IMM 362
November 1967

Courant Institute of
Mathematical Sciences

Programming Considerations for Parallel Computers

E. Draughon, R. Grishman,
J. Schwartz, and A. Stein

Prepared under Grant No. NSF-GP-5673
with the National Science Foundation

New York University

IMM 362
November 1967

New York University

Courant Institute of Mathematical Sciences

PROGRAMMING CONSIDERATIONS FOR PARALLEL COMPUTERS

E. Draughon, R. Grishman, J. Schwartz, and A. Stein

The research carried on to produce this report was sponsored by the National Science Foundation, Grant No. NSF-GP-5673.

TABLE OF CONTENTS

PROGRAMMING CONSIDERATIONS FOR PARALLEL COMPUTERS

by

E. Draughon, R. Grishman, J. Schwartz, and A. Stein
Courant Institute of Mathematical Sciences, New York University

1. Introduction

The present article reports on some investigations, concerning software for a class of hypothetical highly parallel computers, which have been conducted during the past year at the Courant Institute. The general class of computer to which these investigations apply, called Athene class computers in a previous article [1], is characterized by the following properties:

(a) several processing units share a common memory;

(b) each processing unit has its own instruction location counter, and may therefore execute unconditional or conditional transfers independently of all the other processors;

(c) processors may execute code sequences located in their common memory;

(d) the processors comprising the complete system operate asynchronously, that is, in no close time relationship to each other;

(e) each processor may execute all the instructions of a reasonably adequate machine instruction set. Each processor

has its own upper and lower memory limit registers, providing memory protection for a multiprogramming operation system. In addition, each processor may execute the instructions belonging to a small set of special "collective" or "coordinating" instructions (cf. below for additional details).

A simulator for a hypothetical machine of this sort has been written to run on the CDC 6600 presently at the Courant Institute. This simulator, called SOAPSUDS, is adapted from an autosimulator developed previously for the CDC 6600. It simulates up to 60 processors, each with the instruction set of the 6600, sharing a common memory. In addition, the simulator incorporates various coordinating instructions (the precise set included has varied through the course of our experiments). Finally, various aids to debugging and simulated performance measurement are included in the simulator.

In the course of our experimentation with and consideration of the SOAPSUDS parallel computer simulator and associated systems, we have gathered information concerning both the theoretical and practical aspects of writing parallel programs.

In what follows we shall discuss the various questions having to do with storage usage, program flow organization, system efficiency, methods that have been developed to handle parallel programming by normal Fortran techniques, and a new parallel oriented Fortran (PFORTRAN) incorporating these techniques.

A first appendix will give additional information concerning the PFORTRAN compiler. A second appendix will describe a hardware compiler-simulator developed to make future detailed studies of logic design possible.

2. SOAPSUDS: A description of the simulated machine and the simulator.

Soapsuds is a program written for the CDC 6600 to simulate a computer system consisting of a set of central processing units, each with its own instruction location counter, operating asynchronously, from a common memory. Each CPU has an instruction set adequate for a general purpose computer, and can execute unconditional and conditional branches independently of the other processors. The large central memory is also divided into independent modules, the processors and memory modules interconnected in a "central exchange."

The individual processors of the simulated system are essentially the same as the 6600 central processor, with a 60-bit word length and 24 operating registers. This, of course, makes simulator coding and usage much easier. In addition, the large word size and register-oriented instruction set reduce the rate of memory references, making this processor suitable for a system in which memory speed and memory-processor communication may be major factors in overall speed.

Several instructions have been added to the normal instruction set of the 6600 to meet the needs encountered in a parallel processing environment. Since the processors are identical and operate from a common memory, some simple device enabling a processor to tell "which one it is" appears necessary. A unique identifying number for each processor in the system is needed, for example, within the operating system or other code executed by all processors, in which code tables are partitioned for use by the individual processors. In particular, a hardware identification is necesssry if one processor is able to interrupt other processors, since the specification of which processors are to be interrupted must ultimately be translated into hardware terms. For this reason we have included the instruction READ BADGENUMBER in all our studies of parallel processor simulation. This instruction puts the number (which lies between 0 and n-1 for a set of n processors) of the processor executing the instruction into an index register. Suggestions have been made for an "assigned badgenumber" scheme, which, in contrast to the above (absolute) badgenumber, assigns each processor executing a particular job a unique identifying number, starting with zero. It seems satisfactory at the moment, however, to carry identifiers of this latter type to an ordinary index register.

Three instructions which facilitate coordination of processor activity have been studied. OR, ZERO, AND, SKIP,

OR TO STORAGE, and REPLACE ADD. The OR, ZERO, AND SKIP takes the logical sum of the designated X registers of all processors executing the instruction simultaneously and leaves the result in all these registers, clears the present instruction word to a no-op, and skips to the present instruction location+2. The OR TO STORAGE forms the logical sum of an X register and a memory location, leaving the result in memory. The REPLACE ADD computes the integer sum of a memory location and X register, leaving the result both in memory and in the register; furthermore, no other processor may reference the addressed location between the times one processor accesses the location and the time at which the sum is stored.

The OR, ZERO, AND SKIP (OZS) was used, for example, in programming a fork procedure; the necessary code constituted about 20 instructions. A typical application of the OR TO STORAGE is the assemble or join operation: each processor is assigned a bit in a word, and when all bits have been set, all CPUs have been assembled. Because two processors may OR their bits in before either has an opportunity to read the word, however, there is some difficulty in selecting a "last processor" to execute subsequent code. Various difficulties and inconveniences of this type caused the abandonment of both OZS and ORS, and their replacement with the more useful "replace add" instruction.

The REPLACE ADD instruction is now used to implement all processor-coordinating operations. It permits very efficient fork and join, lock and unlock operations. At the same time it provides, in a single instruction, a basic system operation -- the updating of pointers -- which would otherwise require a lock, update, unlock sequence to prevent two processors from accessing a pointer simultaneously. The REPLACE ADD overcomes the timing difficulties inherent in the OZS by being storage oriented, and other difficulties inherent in the OR TO STORAGE by leaving its result directly in a register. In hardware terms, the instruction would probably require a memory module lock out to prevent further references between the time the original contents is issued to the CPU and the time the sum is returned.

Two additional instructions, EXIT and EXCHANGE EXIT, have been added as being convenient for a multiprogramming operating system. These instructions assume that each CPU has its own status flip-flop, and that depending on the setting of this flip-flop the CPU can be in either "program mode" or "resident mode." The EXIT instruction toggles this flip-flop, and at the same time jumps and resets memory limits of the processor. When EXIT is executed by a processor in program mode, the transfer address and memory limits which result are fixed: they become those of the operating system resident program. When EXIT is executed in resident mode, the resulting

transfer address and memory bounds -- those of the next job to be executed -- may be specified in the EXIT instruction. Using this instruction a CPU can be prevented from a program accidentally interfering in another job or with the system resident.

Using an EXCHANGE EXIT instruction, a processor in resident mode can interrupt any other processor and transfer it to another task at the same time; all the registers of the interrupted processor are saved and new ones loaded from memory. Using this instruction, one CPU can obtain the assistance of as many others as are needed to service a real time interrupt, for example. When real time requirements of this kind have been satisfied, the original registers can be restored and the job on which the interrupted processors were working can be continued.

The SOAPSUDS simulator is an executing simulator for the computer system as described above, and allows up to 60 simulated processors. The simulator processes one instruction per CPU in each of its major phases, cycling through the processors. The simulator is equipped with extensive tracing, trapping and checking features and with various options for the timing of routines and the gathering of useful statistics.

The present simulator includes the READ BADGENUMBER, REPLACE ADD, and EXIT instructions. In addition, recent versions have included a simulated "hardware" implementation of private storage (cf. the next section). Storage of this type has been simply and efficiently implemented in SOAPSUDS using an indirect addressing scheme.

3. FORTRAN-oriented parallel programming methods. Storage allocation.

The availability of a replace-add instruction, and the use of the conventions concerning variable naming and memory allocation to be described in the present section, makes the FORTRAN programming of parallel programs easy.

We distinguish at the FORTRAN level between two types of variables PUBLIC and PRIVATE (as will be seen, this classification cuts across the normal FORTRAN REAL, INTEGER, LOGICAL, etc. types).

Variables declared to be PRIVATE are taken to be private to each processor executing the FORTRAN program. That is, it is assumed either that

(a) such variables are stored in registers private to each processor but not in memory, or

(especially if, as is usual, the number of registers per processor is small)

(b) the machine code compiled from the assumed FORTRAN source is such that every processor loads each such variable from and stores each such variable into a memory cell private to the processor.

Implicit 'temporary storage' locations created by the FORTRAN compiler are assumed to be private to each processor, in the sense explained above.

Variables and arrays declared PUBLIC on the other hand are taken to be common to all processors, so that values of such variables set by any processor will subsequently be used

by any other processor, etc.

Typically input data, output data and intermediate results that are needed at some later time are placed in PUBLIC storage, while loopcounters and temporary intermediate data are kept in PRIVATE storage.

In addition to the two main types of storage described above and directly implemented in the PFORTRAN compiler and the SOAPSUDS simulator, two other secondary storage types may be distinguished by their usage. These secondary storage types have no independent logical existence in either the compiler or the simulator however; they are programmed in terms of PUBLIC and PRIVATE storage using the replace-add instruction in ways that will be explained in subsequent sections of the present report. These two secondary forms of storage are (i) Intermediate storage, which is storage from portions of which all other CPUs will be temporarily excluded by some particular CPU. Intermediate storage is typically used for incomplete partial results, for the updating of arrays, etc. (ii) Control storage is used for communication between the CPUs; it is PUBLIC except during modification by one of the CPUs. Such modification is generally of such short duration as not to warrant the use of machinery required for the implementation of storage. Control storage is currently implemented very directly in terms of the replace-add instruction.

4. FORTRAN oriented parallel programming methods.

The basic NEWVAL function.

The 'replace add' machine instruction may be reached conveniently from FORTRAN source programs by introducing a single FORTRAN integer function NEWVAL(I,J). This function has the value I+J. Moreover, each time it is called, it changes the value of the variable I to I+J. If several processors call this function simultaneously, the effect is the same as if these processors called the function in some serial order.

Using this function and the storage conventions explained above has the following consequences:

(a) FORTRAN statement sequences may be executed by several processors at once without unanticipated interference. Consider, e.g., a typical FORTRAN sequence such as

```
      DØ 1 J = 1,10
    1 A(I,J) = A(I,J) + B(I,J) * C(I,J)
```

Assume for the sake of discussion that the arrays A, B, C have been declared to be PUBLIC, but that the integer variables I and J have been declared PRIVATE. Then

(1) During execution of the DO loop each processor has its own 'private' value of the DO counter J; these values are separately incremented. Each processor may also be executing the loop with its own individual value of I.

(2) However, the arrays A, B, C referenced by all processors are the same. Since I and J are particular to a given processor, it may nevertheless be true that the processors are always referencing non-overlapping sets of array elements.

(b) The basic function NEWVAL can be used to express various of the 'FORK' and 'JOIN' instructions previously proposed as fundamental terms for parallel coding (cf. [2] for such a proposal). To have one processor branch to label 1 while all others continue in sequence one uses a COMMON variable I initialized to -1 and the statement

```
IF(NEWVAL(I,1)) GØ TØ 1
```

To cause all processors to go to label 1 when the statement 2 has been executed M times or more (JOIN instruction, cf. [2]) the statement

```
2    IF(NEWVAL(I,1) .GT. M) GØ TØ 1
```

may be used.

A PUBLIC variable I may be reserved for temporary use by a single processor as follows. With I, associate an additional PUBLIC variable, called, e.g., ILOCK, which when I is not in use, is set to -1. The appropriate code sequence is then

```
100  IF(NEWVAL(ILØCK,1)) GØ TØ 100
     ...
     [Here follows code making exclusive
      use of I]
     ILØCK = -1
```

The last statement releases I for use by a processor waiting at,

or subsequently entering, the 'reservation statement' 100.

(c) Parallel DO loops, that is, loops enclosing code which is to be executed a fixed number N of times, may be written as follows. (In the code sequence below, we assume that the DO loop increment is 1, that I and LIM are PUBLIC variables, and that I is set to 1 less than the desired loop initiation value when the loop is entered, and that II is a private variable. Of course, these assumptions are readily relaxed.)

```
300   II = NEWVAL(I,1)
      IF(II .GT. LIM) GØ TØ 200
      ...
      [Body of loop;  II occurs as a variable]
      GØ TØ 300
200   CØNTINUE
```

5. FORTRAN oriented parallel programming methods. Verbs added to the PFORTRAN compiler.

While all necessary code for the coordination of parallel programs can be written directly in terms of the basic NEWVAL function, our programming experiments indicated the utility of providing a more convenient and flexible method for writing parallel programs. In order to experiment with alternative parallel oriented compiler verbs, a Fortran-like compiler PFORTRAN was written, the current specifications of which follow.

(i) The basic features of Fortran IV are all available.

(ii) Storage types.

The compiler generates code for two storage types: PRIVATE and PUBLIC. Private storage is implicitly indexed by a processor badgenumber, and therefore provides each operating CPU with its own individual value of any particular PRIVATE variable. PRIVATE storage is implemented by reserving a unique area for each given private variable and treating all references to that variable as a reference to the head of the area plus the CPU number.*

Public storage is handled in the more familiar manner of ordinary FORTRAN.

(iii) Looping.

It is useful to define two new forms of parallel DO statements and to note a modification in the meaning of the ordinary DO statement.

Our first new 'parallel DO' statement form is

(a) DØP S(A) J = IN, FIN, ST

Here S and A are labels. DOP starts the CPUs executing it with consecutive initial loop-count values (i.e. J=IN,IN+ST...). Any CPU whose assigned initial value J exceeds INT is shunted off to statement A; one CPU is allowed to fall through statement S.

The second new 'parallel DO' statement form is

(b) DØP S J = IN, FIN, ST

This form of DOP operates exactly like the other except that

*Every CPU is given a unique number; this identifier is generally held in internal index register A0.

when the assigned initial value J of a CPU exceeds FIN, control of any CPU whose assigned initial value J exceeds FIN is transferred to the first statement past the loop.

Note that execution of a statement of either of these forms sets up a counter at the head of the loop. The counter is decremented at the close, in order to allow the proper termination of the loop.

Note also that when an ordinary DO is used:

(a) if the DO variable is PUBLIC, care must be taken in order that only one CPU executes the loop at one time.

(b) if the DO variable is PRIVATE, the loop may be executed simultaneously and independently by any number of CPUs. Such execution causes each of the CPUs to execute the loop with the same range of values given to the variable. As the loops are not coordinated, explicit coordination must be implemented elsewhere in the program as necessary.

(iv) Parallel execution control statements.

The PFORTRAN language includes several verbs which enable the user to coordinate the parallel execution of portions of code in a convenient manner. These statements are LOCK, UNLOCK, LOCK(K), UNLOCK(K) and PAR N,S.

LOCK allows only one CPU to enter the code which follows it. UNLOCK opens the lock appearing most recently in the program. Unlabeled LOCK and UNLOCK statements are thus required to form a nested set.

LOCK(K) operates much as the simple form of the LOCK statement. Since a parameter is explicitly provided, however, 'unlocking' from a remote point of the program is possible.

UNLOCK(K) has a self-evident meaning.

PAR N,S allows not more than N CPUs to enter the following program; if more than N processors execute the PAR statement, control of the N+1-st and N+2-nd, etc., processors is transferred to statement S.

UNLOCK S resets the PAR filter appearing as statement S.

(v) The basic NEWVAL function, whose meaning is explained in the preceding section, is provided as a built-in function of the PFORTRAN language.

(vi) Two system verbs are provided to enable PFORTRAN user programs to communicate with the simulated operating system described in the following section. Within this simulated operating system, compiled programs will be operated in a time-sharing environment. They will have the ability to request CPUs from and to return CPUs to the system. The two verbs that facilitate this user-system communication are REQUEST and RELEASE.

REQUEST has the form

$$\text{REQUEST } N_1, ST_1, N_2, ST_2, \ldots$$

where the N_i are integers and the ST_i are statement numbers. When this verb is executed, the system is requested to provide N_1 CPUs to begin execution at ST_1; N_2 to begin execution at ST_2, etc.

The RELEASE statement has the form

RELEASE

Any CPU executing this verb will return to the operation system.

6. The simulated ATHENE operating system. Outline.

The purpose of that portion of the operating system which has been written so far is (1) to re-distribute processors, as they become available, to various tasks attempting to run in the simulated operating environment; (2) conversely to handle the assignment of tasks to processors; and (3) to accept and manage the I/O for all processors.

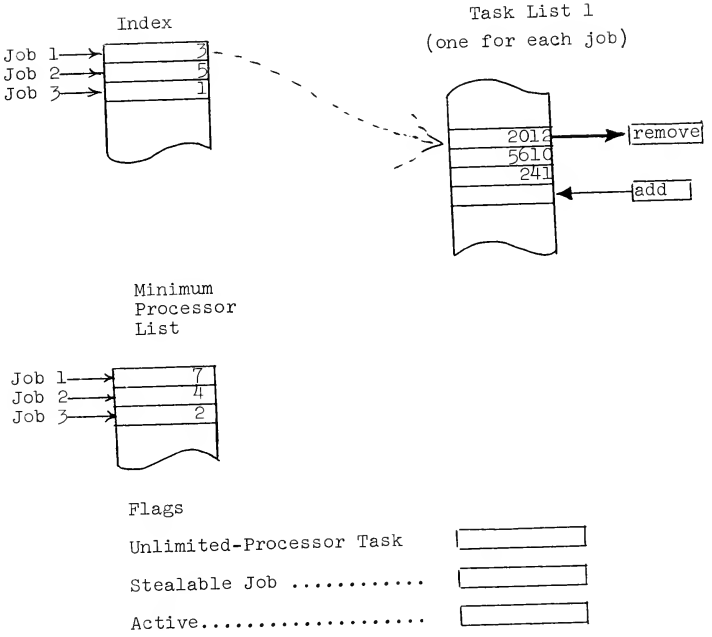
Executive control resides temporarily with any CPU executing a portion of the resident program itself, rather than permanently or semi-permanently with a particular CPU. Any number of CPUs may be executing portions of the resident program at the same time; little or no waiting for other CPUs to terminate will occur.

The tables used are a 'joblist', and a 'tasklist' for each job in the joblist (see Figure 1). The 'joblist' contains a slot for each job currently in the machine, and each slot contains a subfield giving the total number of tasks currently available for that job. As tasks are added and removed, these totals are incremented and decremented via the RAD instruction which avoids unnecessary waiting almost completely. Each 'tasklist' describes all the unassigned parallel tasks for a

ATHENE

Parallel Processor Operating System

Control Tables



given job on the joblist, and contains an address for each task currently available for execution. When a task is picked up by a CPU, it is removed from its tasklist. Additions to tasklists are caused by processors assigned to a given job executing REQUEST verbs. There are separate pointers for each of these tasklists, indicating a next task addition and a next task removal address. Both addition and removal can be done concurrently with no waiting time.

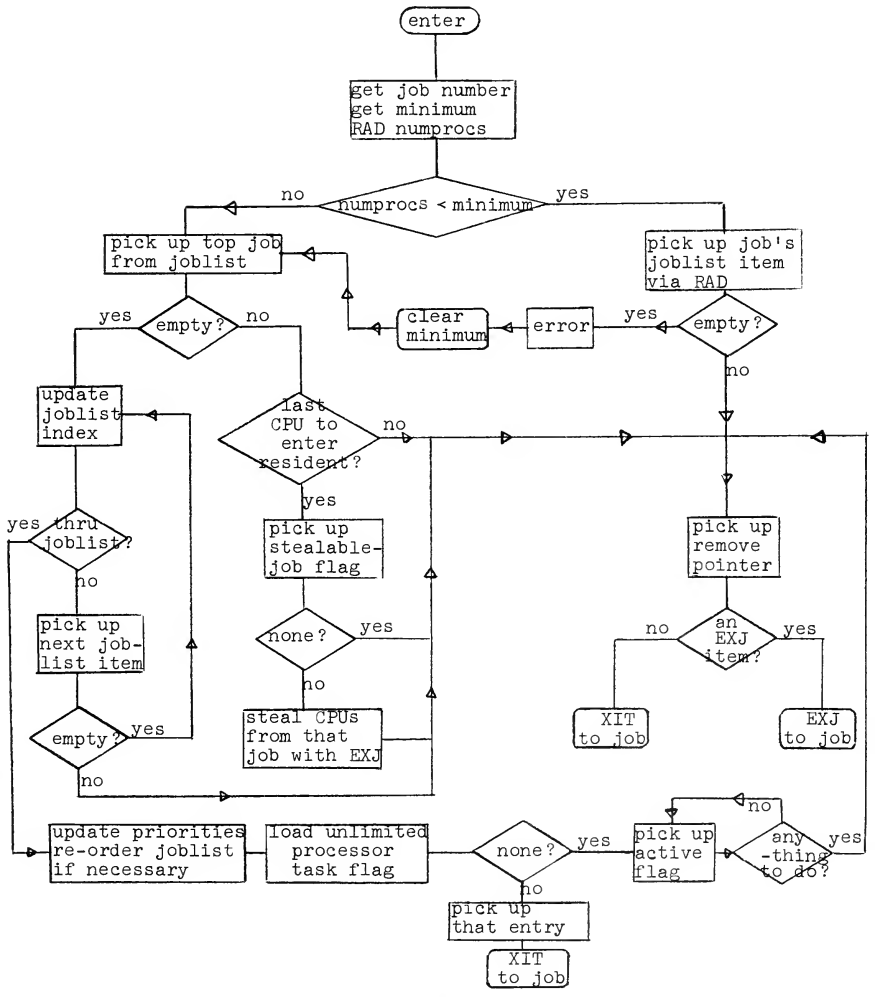
The method employed may be described as follows. (See flow chart, Figure 2.) (1) Each processor, on executing a RELEASE, checks that its exit from a job has not reduced the number of processors for that job to below the minimum assigned by the user. If this is not the case the processor finds the joblist slot of the highest priority job and decrements the total count of the number of tasks available within this job by 1, using the RAD machine instruction. The CPU then picks up the remove pointer for the appropriate task list via the RAD and EXITs to that task. Note that even if two or more CPUs execute this section of the executive code simultaneously, no waiting time will be forced, and each CPU will get a unique task from the list. The tasklist for each job is circular and is (arbitrarily) 100 octal words long. The tasklist pointer, after being incremented by the RAD instruction, is masked with a 77, thus obtaining circularity without any delay for resetting the pointer every time it reaches the end of the list.

If no limited CPU-number job requires any additional

Athene

Parallel Processing Operating System

Algorithm



processors, a processor entering the system may either execute the system job rollin-rollout code, or it may go to a task that can accept an indefinite number of CPUs.

In real-time emergencies, or for efficiency, or when a job aborts, it may be necessary for a processor to steal other processors from a job via the forced EXCHANGE EXIT machine instruction. Aside from housekeeping, there are no differences between tasks to be entered with an EXJ and those entered with a normal XIT.

(2) To execute the REQUEST verb, a processor EXITs to the resident code, and increments its job's tasklist add pointer, using the RAD instruction, a number of times equal to the number of tasks it is going to add to the tasklist. It then stores the addresses of the tasks on the tasklist list. Other additions to the list can be going on concurrently and no waiting time is forced.

(3) The user assigns a run-invariant output grouping control type to his job, to control the arrangement of his output. Output may be grouped by processor, by task, or chronologically, the necessary arrangement of output files being accomplished by the system code.

All output is transferred unconverted to one of two dual buffers. When a processor finds a buffer full, it begins converting its contents and sending them to the system I/O control processors for physical output. Thus, only one

processor is required for output conversion and I/O processing. Other job-attached CPUs are freed immediately for further computation. The likelihood of waiting forced during the dumping of buffers is thereby reduced relative to what would have been expected if separate buffers for each job or even for each task or processor had been maintained.

7. The simulated ATHENE operating system. General discussion of operating system problems.

Any operating system for Athene-type parallel machines must solve the following problems:

(a) how to route the CPUs between several jobs in a minimum of time.

(b) how to establish a job mix that keeps all CPUs occupied.

(c) how to define and implement priority.

(d) how to optimize throughput.

First, as to throughput. It seems advisable to guarantee each job a minimum number of CPUs, to let the job itself specify what this minimum shall be, and to allow this minimum to be changed dynamically.

In the present system, the minimum may be changed each time a request for processors is made. The minimum is checked each time a processor is returned to the system. If the minimum is too high relative to the other jobs, the job is rolled out.

Second, as to useful techniques for CPU routing. Two methods have been studied and coded. Method 1: maintain a queue of tasks for each job in the machine. A pointer to this list (both the queue and the pointer are public variables) is picked up via the RAD instruction by the individual processors acting independently. Each processor is thus assured a unique task from the list. In adding tasks to this list (simultaneously of course), a processor uses another pointer which it updates by an amount equal to the number of tasks being added (using the RAD instruction). The list is circular and overflow is detected by noticing whether the upcoming slot is zero or not. In addition, an index of jobs is kept (and ordered by priority). On this index one maintains the total number of tasks available on the corresponding tasklist. Maintenance of this total without waits or lockouts is accomplished by the RAD instruction.

Method 1, as described so far, involves no waiting. In an earlier version, it was necessary for the processors to wait whenever the pointers were reset to the top of the tasklist list so that the list could be used circularly. In the present version, however, after the execution of a RAD instruction, the right-most six bits are masked out (since the list is 100 octal words long) and these six bits provide the pointer. The circularity is thus automatic.

When priorities are changed, however, or when new jobs are rolled in, the master index must be re-ordered by priority,

and this involves waiting time.

To deal with this problem a second method was devised (Method 2). In this method, separate lists are maintained for each priority level. The lists operate as in method 1, except that tasks are added to the list corresponding to the job's current priority. It is thus easy to change priorities and there is no necessity ever to re-order a list.

The difficulty with this method is that when a job aborts or is rolled out temporarily, it is especially tedious to search for and remove all the outstanding tasks from the various lists on which they may appear.

A third general desideratum is keeping the machine busy. Some algorithms can use a large variable number of CPUs, so that the number which may usefully be assigned to such a job is for practical purposes unlimited. If there is at least one job in the machine with such an unlimited-processor task, then all the CPUs will be busy. It is thus important to know which jobs have such tasks available and when they will be available. In this connection it is important to know how plentifully such algorithms will occur in a normal installation job-stream.

A number of problems are common to both of these methods.

As the number of jobs and the number of CPUs increases, the housekeeping problems involved in storage and retrieval of the exchange packages increases. On the other hand, it is not clear that EXCHANGE EXITs are needed except in cases of aborts

or roll-outs. (No EXCHANGE EXITs have been used so far in the present systems.) Priority considerations, however, may require that a processor should steal other CPUs via the EXCHANGE EXIT instruction. This may be done when the queue for a high-priority job becomes too long, or when a low-priority job has accumulated more CPUs than it should have, given its priority. A difficulty arises in this connection in the use of unlimited-processor tasks. If these tasks are long ones, it may be more appropriate for some other task to obtain the processors so that reassignment would be indicated. However, it is possible that the same processor may work on more than one task within the same job. If this processor is interrupted overfrequently, then the number of exchange packages per job may exceed the number of CPUs in the machine. Yet, priority considerations do not seem to allow one to require that a processor come back to take up the interrupted task before doing anything else.

No provision has yet been made in this system for cancelling tasks, though of course it is desirable for the user to be able to request the exeuction of a task and later withdraw it, either before it has begun execution or afterward.

If a group number register were included in the machine hardware, it would be possible for a number of processors to be re-apportioned simultaneously. Suppose all the processors on one job or task are assigned the same group number. A system CPU could join that group and jump all processors of that

group to a common address (this jump would be like a return jump). At that address, the processors, using the RAD instruction, could split themselves into various streams, so that all but a certain number could return to where they were; the rest could return to the system to find out what tasks they are scheduled to perform.

A number of questions arise in connection with output in a parallel environment:

(a) should there be standard alternative ways of organizing the output from a given job;

(b) should outputting (printing) from public variables be allowed;

(c) should the data be converted before it is put into the output buffer by the individual processors, or afterward;

(d) should the type of organization of the output be run-invariant or be changeable by the programmer;

(e) should non-standard types of organization be possible to the programmer?

In the present system, there are three standard types of organization of the output: (1) by processor, (2) by task, (3) by chronological order. The type of organization selected by the user is run-invariant.

Instead of having separate output buffers for each processor or for each task, it was decided to have one job master buffer. Some of the reasons for this are: a master

buffer would waste less space. Moreover, providing an output buffer for each task within a job would create substantial delays while the various buffers were being dumped. More than one CPU would have to wait for I/O controller to become available to do the physical output.

No provisions for non-standard types of organization have been made in the present system.

Output from public variables seems somewhat problematic. As far as FORTRAN is concerned, it is not possible to indicate to the programmer that once one processor begins to execute a PRINT statement, it may not be safe to use the public variables in the FORTRAN I/O list until the entire PRINT statement has been completed. Such a convention would result in unnecessary waiting time (especially if the data is converted at that time). Yet since it is sometimes necessary to print out public arrays, it seems best to leave the decision explicitly to the programmer.

8. Effective organization of parallel program flow.

In order to make effective use of available computer power, programs that are to be 'parallelized' should be set up so that only a minimum of time is lost waiting for intermediate results. If waiting processors are to be RELEASEd and then reREQUESTed, the waiting periods should consist as much as possible of a small number of long gaps rather than a large number of short ones, to minimize the costs inherent in switching CPUs from job to job.

To accomplish these goals, it is useful to do a preliminary analysis of the Pert type (either by hand or computationally) of a program to be made parallel. The results of such an analysis would describe the following aspects of the overall program structure:

(1) dependencies of significant computations on previous results;

(2) portions of the program that can be done in parallel;

(3) maximum number of CPUs that can be usefully employed at each stage of a program run;

(4) an estimate of execution time for the various program segments.

This information can then be used in designing an efficient parallel program layout by a judicious placement of LOCKs, UNLOCKs, PARs, REQUESTs, and RELEASEs, in accord with the following set of rules:

(i) REQUEST the maximum number of CPUs available for those portions of the critical computation path that can be performed in parallel.

(ii) Within program segments that are of relatively long duration and lie on the critical path, use a combination of PAR and RELEASE to return the idle CPUs to the operating system.

(iii) Within critical program segments of short duration use LOCK and/or PAR to temporarily idle the unnecessary CPUs.

(iv) Subtasks that do not lie on the critical path do not justify

the use of more than one CPU, so that CPUs beyond the first should be LOCKed out of these subtasks.

(v) The task assignment structure should be so set up as to allow CPUs that have finished one particular subtask to start on a concurrent task that has not yet been completed.

A rudimentary automatic 'parallelization' is currently being planned. This mechanism will examine DO loops for inherent parallelism and arrange for the DO loop of largest possible range within each set of nested DOs to be done in parallel. The compiler will examine the body of each DO loop to determine whether at each stage in its iterative execution currently computed values are independent of prior traversals of the loop. If this is found to be the case, search continues to the containing DO and so forth. On completion of the loop analysis, the compiler will generate an appropriate REQUEST followed by a DOP. A RELEASE of all but one CPU will be placed by the compiler at the end of the loop.

9. A qualitative account of various parallel programming experiments.

The few modifications to Fortran mentioned in Section 5 above have the fortunate consequence that a large variety of available Fortran programs can be reprogrammed to run in parallel with a minimum of effort. We have programmed a variety of problems using this technique. A sampling of the

programs considered, with brief comments on each, follows.

(a) Matrix multiply. Made parallel by assigning each available processor a row. Trivial, by conversion of outer DO loop of standard procedure to parallel DO loop.

(b) Internal sort of array. Performed by insertion of successive elements into pre-sorted set of elements, using binary search. (This sort requires no supplementary storage.) Made parallel by making the string a two dimensional array, giving each processor a row to sort and varying the dimensions so that the rows being sorted increase in length.

(c) Parallel minimax search (Kaplus-Miranker algorithm, cf. [3]) for locating maximum of a function. Each processor was assigned a point for the computation of the function in each of the successively smaller intervals established in the algorithm.

(d) 'Parallel shooting' method for solving two-point boundary value problem for ordinary differential equations (algorithm due to H. B. Keller). Made parallel by assigning a processor to the computations of each subinterval into which the full interval of the problem is divided.

(e) Calculation of eigenvalues of complex matrix by the QR method. This program was developed by adaptation from a complex eigenvalue library routine (QREIGEN) used at the Courant Institute Computing Center. It was made parallel by assigning row and column operations to individual processors and by the parallel

computation of the components of sums and products.

(f) Monte Carlo programs for atomic energy level calculations. These programs, developed recently in connection with other research efforts at our Computing Center by Dr. M. Kalos, lent themselves very readily to parallel treatment. For these programs it was arranged that each processor be given a unique set of random numbers and the computations and list building are done completely in parallel. Existing FORTRAN programs whose underlying logic makes them amenable to parallel treatment may readily be converted to parallel programs; this conversion is not substantially more difficult than any other sort of program modification.

Our experience may be summarized as follows.

(a) Parallel programs may be written in the modified FORTRAN described above. The resulting programs are similar in appearance to, and often not significantly more complex than, single processor programs accomplishing the same calculations.

(b) Programs in which the bulk of processor time is spent in a DO loop which may be performed in parallel are of not infrequent occurrence. These may be converted to run in parallel with good efficiency by the trivial expedient of changing appropriate serial DO loops to parallel DO loops. This may be done either by giving unique loop variable values to each executing processor, in those cases where the computation is carried out with different values of the variable; alternatively,

this may be done by simultaneously varying the loop variable in those cases where computations are carried on in parallel over the same variable range, as in the sort, the Monte Carlo computation, and the parallel shooting calculation mentioned above. Finally, we note that DO loops may be made parallel by any combination of these two methods.

We make one additional remark on simulation-compilation technique. Since the PFORTRAN compiler has not been available for our early parallel programming efforts, a provisional scheme based on the use of the existing 6600 FORTRAN compiler and on a simple modification of the SOAPSUDS simulator was employed. In this provisional technique, private storage is provided for the compiled programs in terms of the following convention: variables and arrays occurring in FORTRAN programs, and not assigned to FORTRAN 'numbered COMMON', are taken to be PRIVATE to each processor executing the FORTRAN statement. On the other hand, variables and arrays assigned to FORTRAN 'numbered COMMON' are taken to be PUBLIC to all processors, so that values of such variables set by any processor will subsequently be used by any other processor.

Implicit 'temporary storage' locations created by the FORTRAN compiler are private to each processor, in the sense explained above.

Intermediate storage is simulated by locking out more than one CPU from the portions of the program in which intermediate

variables are computed; the locks are written directly in terms of the basic NEWVAL function.

10. Quantitative data on computational efficiency for a sampling of parallel programs.

Following is a set of charts which summarize our actual running experience. Along with the actual observed running time vs. number of CPUs, we have plotted an efficiency measure which we have taken to be

$$E_N = \frac{T_1}{N * T_N}$$

E_N = efficiency for N CPUs

T_1 = time required for a serial machine to perform the program

N = number of CPUs

T_N = time required for N CPUs to perform the program.

(The efficiency as here defined may be greater than one in some applications e.g. parallel searches, theorem provers, etc.)

Complex Eigenvalue Problem where the program and data configuration was such that more than 4 CPUs were useless.

The interference effects of the unnecessary CPUs is quite apparent. They are caused by delays built into the program that were used to organize the CPUs.

Shockwave Problem cut off at end of (essentially) parallel computation.

The initial drop in efficiency was caused by the parallel coordination coding.

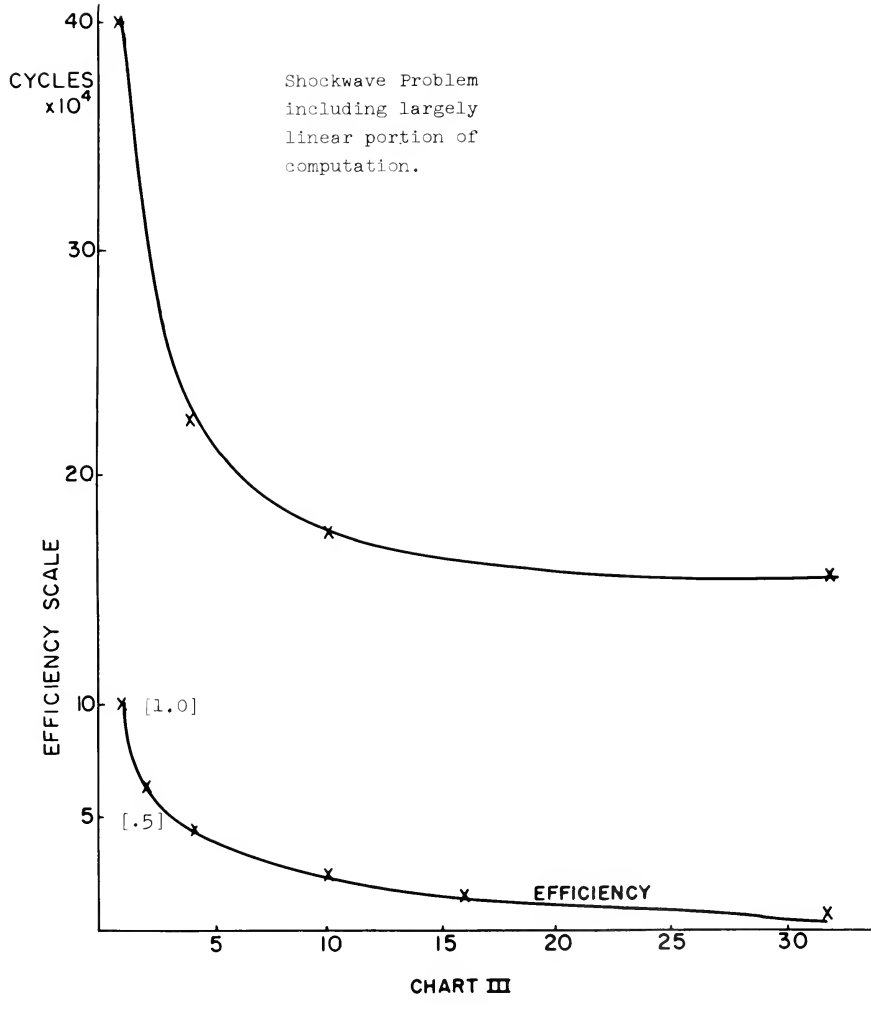

40
CYCLES
x10^4
30
20
10
5
EFFICIENCY SCALE
Shockwave Problem
including largely
linear portion of
computation.
[1.0]
[.5]
EFFICIENCY
5
10
15
20
25
30

CHART III

Monte Carlo Computation (with queued random number generator).
The frequent executions of the random number generator required by the algorithm caused the increase of running time when more than eleven CPUs were used. (This could be remedied by assigning each CPU a different base number set up in such a way so as to insure independence of the random variables thereby eliminating the necessity for the queue.) Note, that for less than six CPUs the queue effect is not noticeable.

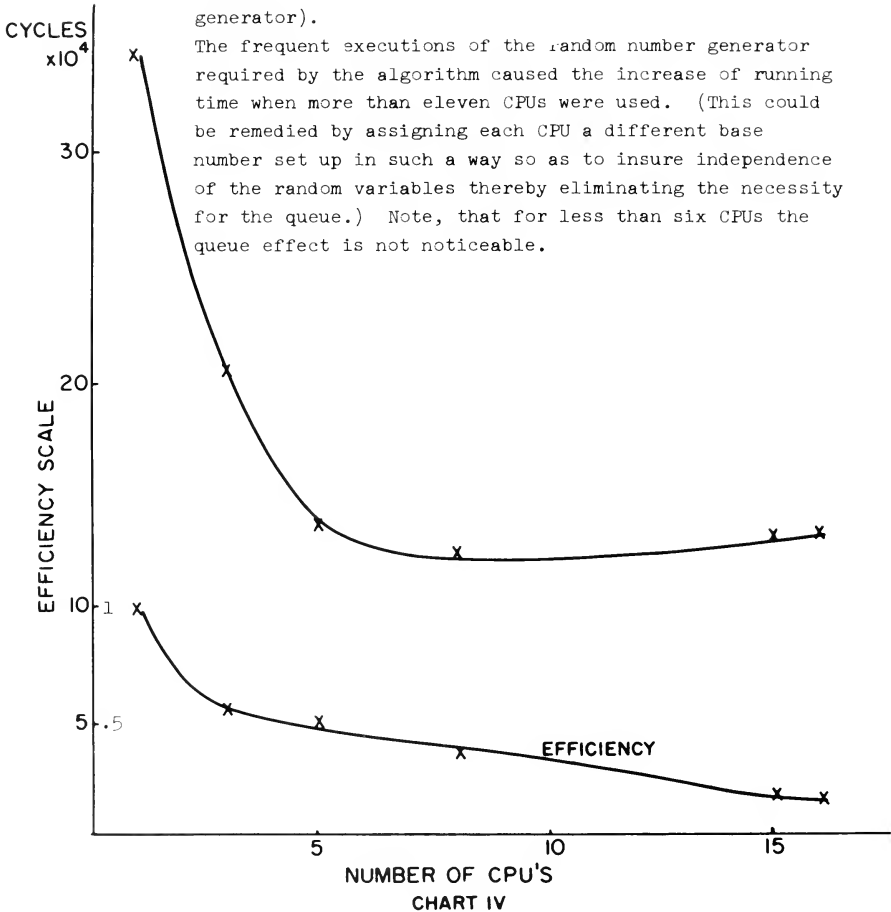
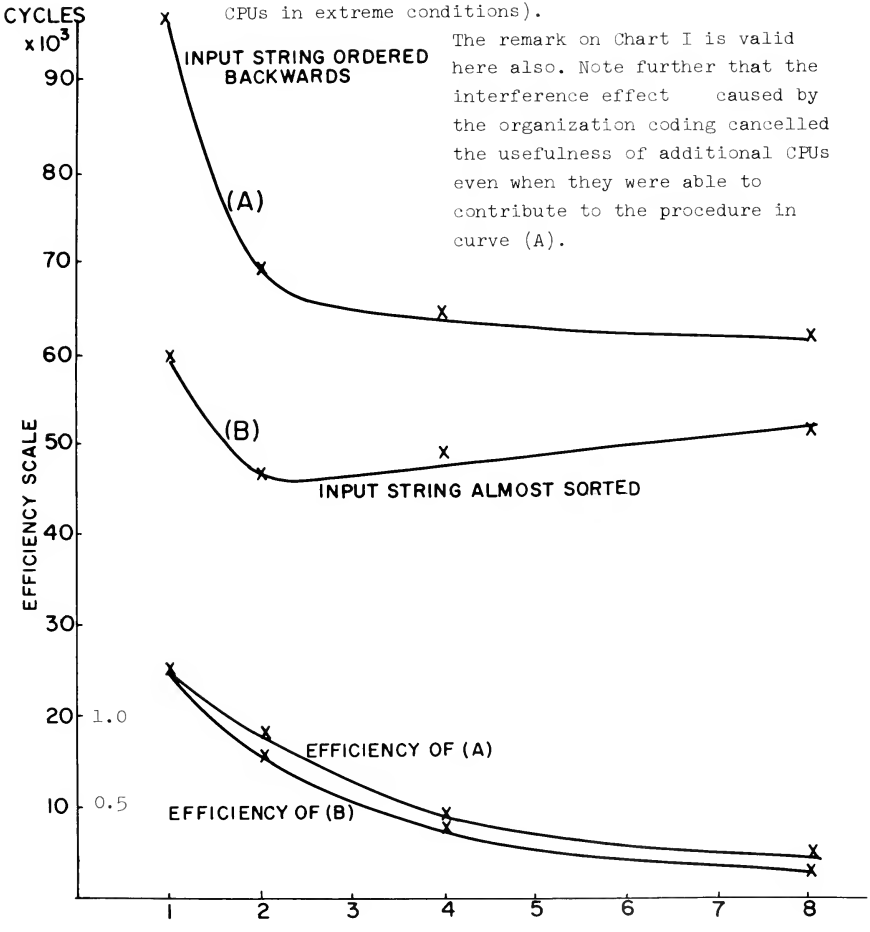


CHART IV

Parallel Sort (showing the effect of additional CPUs in extreme conditions).



The remark on Chart I is valid here also. Note further that the interference effect caused by the organization coding cancelled the usefulness of additional CPUs even when they were able to contribute to the procedure in curve (A).

CHART V

Operational Statistics on a Large Number

of Runs of the Monte Carlo Problem

Table I

Number of CPUs	Useful Time		Parallel Proc. Overhead		Time Available for Concurrent Jobs	
	Parallel	Series	Range	Total	Range	Total
3	122.4×10^3	5.7×10^3	$.4\text{-}1.3\times10^3$	1.7×10^3	4.4×10^3 -5.3×10^3	9.7×10^3
3	135.3	5.7	.4-0.9	1.3	4.8-5.3	10.1
3	13.2	5.7	.2-.5	.7	5.2-5.7	10.7
3	103.9	5.7	.7-.9	1.6	4.8-5.0	9.8
3	117.1	5.7	.3-.5	.8	5.2-5.4	10.6
3	144.9	59.2	1.7	3.4	57.5	115.0
8	79.2	5.7	.1-2.4	6.4	3.4-5.7	133.5
8	64.8	5.7	.0-.1	.5	5.6-5.7	39.4
16	78.3	5.7	.1-3.1	21.6	2.6-5.6	69.6
16	71.2	5.7	.1-3.7	19.5	2.0-5.6	71.7
16	79.6	5.7	.2-4.3	19.5	1.4-5.5	71.7
16	92.5	5.7	.0-3.5	16.3	2.7-5.7	74.9
11	78.3	61.5	.1-19.5	35.6	32.1-51.4	479.4
5	85.3	5.7	.0-1.3	2.6	4.4-5.7	20.2
5	69.5	53.9	.2-3.3	5.5	50.6-53.7	210.1

Useful time: The time spent in actual execution of the algorithm (the sub-headings represent the time spent in parallel and series computation, respectively).

Overhead: The time uselessly spent in computation by processors about to be released, before they were able to execute a 'return to system' instruction.

Time Available for Sharing: The time in which the processors executing the algorithm could have usefully been doing other procedures in a time-sharing environment.

This graph represents the ratio of overhead/CPU to useful time vs. number of CPUs for the random (Monte Carlo) runs summarized in Table I. (Note the almost linear relationship.)
%
OVERHEAD / USEFUL TIME
2.2
1.8
1.4
1.0
.6
.2
2
4
6
8
10
12
14
16
NUMBER OF CPU'S
CHART VI

APPENDIX I

TRANSLATION TECHNIQUES EMPLOYED IN THE PFORTRAN COMPILER.

1. The general strategy employed in programming the PFORTRAN compiler.

The source language to be analyzed by a compiler written in the general style which we have employed may initially be described syntactically in a notation very much like the standard Backus normal form. Such a description implies a recursive recognition procedure. This recursive procedure is programmed in detail by the manual transcription of the syntactic description into a set of calls on a recursive family of procedures, described in more detail in the two following sections. When a recognition procedure is recursively called, it may in turn invoke other recognition procedures, as analysis of the source string syntax progresses. Each of the recognition procedures will eventually return either successfully or unsuccessfully. By standard convention, a recognition procedure is provided with two transfer labels as arguments. Control will pass to one argument on successful return, to the other argument on unsuccessful return. Alternatively, if the successful return argument is zero, control will pass, in case of success, to the immediately following recognition procedure call; if the unsuccessful return argument is zero, control will be returned, on an unsuccessful return from a lower

subroutine level, to the next higher level recognition subroutine, the indication of failure being carried along recursively.

The data structure which carries all the necessary return addresses for this recursive action is a simple pushdown stack, called RECRLIS in the PFORTRAN compiler. Two basic recursive recognition procedure controllers are provided: RECR and PROD. The second resembles the first, but includes a mechanism for iteration of a procedure, success or failure of the iteration depending on the number of successful returns made from a lower level.

When the recursive recognition process reaches the 'atomic' level, as eventually it must, one or both of two additional types of procedures, examination procesures and/or generation procedures, will be employed. An examination procedure examines an atom either for identity with a keyword of the language or for membership in some lexical class (such as language keyword, integer constant, Hollerith constant, integer variable, real variable) etc. constituting an atomic type within the language. Normally, an examination procedure merely returns an indication of success or of failure. However, one particular examination procedure, RFIND, which bears the responsibility for recognizing variable subroutine and function names, as well as labels and integer constants, i.e. for recognizing all semantic entities which are not keywords of the language, carries out some

significant additional actions. This routine examines the master hash table (called VARLIST) for prior occurrence of the variable name. If the name does not occur, it is entered into the hash table. Moreover, and perhaps more significantly, RFIND places any element which it recognizes on the top of an auxiliary pushdown stack (PDL) which constitutes the main compiler data structure for the transmission of arguments between recursive routines. Once placed on this stack, such an element is available for subsequent use during the translation process.

Generator procedures modify one or another of the global data tables used in the compilation process. These tables include the master symbol hash table (VARLIST), the recursive subroutine argument pushdown stack (PDL), and the pending DO loop end label table (IDOLIST), as well as other related tables containing auxiliary information, and described in more detail somewhat below.

Among the principal generator routines are ARITH, which generates code for logical and arithmetic statement evaluation; RCALL, which generates code for function calls; DODO, which stacks DO labels on the pending end-label list (IDOLIST); and RLAB, which generates the necessary code for DO loop endings and which checks the well-formedness of DO nesting.

ARITH acts as follows. It is supplied with the number (1 or 2) of its operands, and with an indicator of the particular

arithmetic or logical operation for which code is to be generated. It then finds its operands (including information on operand type, etc.) on the top of the PDL stack, generates code for the required operation into the output code string, and puts a designator for the location of its result (core location or register) back on the top of the PDL stack.

Note that in this arithmetic expression compilation technique, only operands and not operators are stacked on PDL. Information concerning operations is of course carried implicitly elsewhere, namely, in the master control stack (RECRLIS) of pending recognition subroutine returns.

When the beginning of a nested function call of the form ...F(...,...,...,...) is encountered, the function symbol F is placed on the pushdown stack PDL (by RFIND). When the terminating right parenthesis is encountered, RCALL unloads the PDL pushdown stack, down to the first entry flagged as a function name. The elements encountered above this element on PDL generate the various entries in the target subroutine calling sequence, which is of standard form; the function name, when found on the pushdown list PDL, defines the function type subroutine to which transfer is to be made.

The DO loop head generator routine DODO generates all necessary target code belonging to the beginning of a DO loop, and, in addition, stacks the DO label on the top of the DO stack (IDOLIST). The generator routine RLAB, which is invoked when

the translation of a labeled statement (with label L) has been completed, examines the label for identity with the label L stacked at the top of the DO stack. If such identity is detected, the DO stack is pushed down, and target code for a DO loop end generated. This process repeats as often as necessary. If any labels identical to L are found on the DO stack but not at its top, an 'erroneous DO nesting' diagnostic is generated.

The various auxiliary data structures employed in a significant way by the compiler are as follows. A dimension table DIMTAB contains dimension information for arrays; each is referenced by a basic array name in VARLIST. A subroutine argument table JARGLST contains a list of all arguments to the current subroutine, and is used by various generator routines to determine whether 'internal' or 'external' type target code is to be generated when a given variable is to be fetched. This table is cleared whenever an END card is encountered. A statement function argument table ITARLST plays a similar role for arithmetic statement functions. Common block names and other common block information is accumulated in a common block name table COMTAB. Names of called subroutines and functions are accumulated in the subroutine name table SUBLIST. A corresponding local accumulation within the current subroutine is made in the arithmetic statement function table ARTLST. Temporary locations for the storage of variables overflowing

registers are accumulated in a temporaries table ITEMP. Constants and their descriptions are accumulated in two tables ICONS and ICONSM.

Address assignments within a subroutine are made only on encountering the subroutine END card; this allows dimension and other declaration statements to be placed in a subroutine without undue restriction.

Register allocation is accomplished during the compilation process, a single algorithm governing the use of the registers being employed.

2. Additional details concerning the syntactic description employed.

The general syntactical structure of the PFORTRAN language is essentially the same as that of ordinary FORTRAN. The following shows that portion of the syntax which describes the IF statement; we use this particular example to demonstrate our translation techniques. The syntactic description is written in a manner corresponding exactly to its treatment within the PFORTRAN compiler.

```
<IFSTAT> = :    IF <IF>
<IF> = :        (<IFTAIL> | <MISCD>
<IFTAIL> = :    <LØGEXP>) <STØRBR> | <ARITHEXP>)[1] <THRLAB>
                | <IFTAILB>
```

[1] The distinction between ARITHEXP and LØGEXP is designated by switches within the primitive operations.

```
<IFTAILB> = :    <MISC>
<LØGEXP> = :     <CØNJ> P_0^∞ [1]  <LØGEXP1>
<LØGEXP1> = :    .ØR. <CØNJ>
<CØNJ> = :       <LØGTRM> P_0^∞ <CØNJ1>
<CØNJ1> = :      .AND. <LØGTRM>
<LØGTRM> = :     .NØT. <LØGELM> | <LØGTR1>
<LØGTR1> = :     <LØGELM> [2]
<LØGELM> = :     (<LØGEXP>) | <LØGELMA>
<LØGELMA> = :    <RELAT>
<RELAT> = :      <EXPR> | <EXPR> <<RELØP>> [3] <EXPR>
<STØRBR> = :     <TWØLAB> | <STØRBRA>
<TWØLAB> = :     <<STATNØ>>, <<STATNØ>>
<THRLAB> = :     <TWØLAB>, <<STATNØ>>
<STØRBRA> = :    <STATEMENT>
<EXPR> = :       + <TREXP> | - <TREXP> | <TREXP>
<TREXP> = :      <TERM> P_0^∞ <SGNTRM>
<SGNTRM> = :     +<TERM> | - <TERM>
<TERM> = :       <FACTØR> P_0^∞ <OFACTØR>
<OFACTØR> = :    * <FACTØR> | / <FACTØR>
```

---

[1] P_M^N has the meaning: the following item occurs at least M times and as many as N times.

[2] Constructions like these are used occasionally to make the subsequent code generation somewhat easier.

[3] Double brackets will enclose those primitives that are not standard operators.

<FACTØR> = : <BEXP> ** <BEXP> | <BEXP>

<BEXP> = : (<LØGEXP>) | <ARRY> | <<SIMPLE VARIABLE>>

(<PAREXP>) | <<CØNSTANT>>

<ARRY> = : <<ARRAYNAM>> (<SUBSCR>) | <<ARRAYNAM>>

<PAREXP> = : <EXPR> P_0^{∞} , <EXPR>

<SUBSCR> = : <EXPR> P_0^2 , <EXPR>

<MISCD> = : <<ADDUM>>[1]<CMSTL> | <<QUØ>>[2]<CMSTL>

<CMSTL> = : <TWØLAB>

<MISC> = : <<EØF>> <<INTEGER>>) <TWØLAB>

The more detailed form of the control macros used in encoding syntax of the above type into executable form will be explained in the following section. Before doing so, however, let us note the uses intended for our principal control macros.

(a) Simply recursive elements of the syntax (the items enclosed by "< >" in the preceding syntax) are handled by the RECR macro.

(b) Multiply recursive elements (represented in the above syntax listing by items P_M^N followed by a bracketed item) are handled by the PROD macro.

(c) Primitive elements (represented in the above syntax by elements enclosed in double brackets) are detected by the EXAM macro.

(d) Target code generation is handled by the GENR macro.

---

[1] Accumulator overflow.

[2] Quotient overflow.

3. Additional details concerning the control macros used to program the syntactic analysis.

The following set of macros has been found useful, especially for expressing that part of the compiling system which scans the input string (PFORTRAN program) and governs its translation to a completely explicit internal machine form. These macros enable this major portion of a compiler to be programmed by what is essentially a transliteration of the input language syntax as given in a slightly modified Backus notation. Use of this technique makes it relatively easy to change the input language.

The form of the translation control macros which we have employed is as follows.

(i) The RECR macro.

The RECR macro has the form

RECR DEF, FAIL, BACK .

This macro causes a recursive execution of the macro string starting at location DEF; i.e. control is transferred to location DEF and the current location (i.e. the return location) is placed at the top of a return push-down list. FAIL and BACK are two code address pointers, which control the action to be taken on return from a lower level of recursion in the manner explained in what follows.

The macro string at location DEF returns control to the

current location, either on "successful" termination of the macro coded string beginning at DEF, or by the "FAIL" equivalent if the macro programmed code beginning at the label DEF cannot be executed with full success. Note that on execution of the PROD, EXAM, and TEST macros described below, a success-failure bit is set which controls the decision to transfer upon return from a lower recursive level to the BACK or to the FAIL address.

If BACK = 0 in the RECR macro call, execution continues in line on successful return from a lower level of recursion. If BACK = 1 in the macro call, successful return from a lower level will at once force return to the next higher level with the success bit still set.

When FAIL= 0 an unsuccessful return from a lower level of recursion forces return to the next higher level with the success bit reset.

If FAIL is not zero, and control is returned from a lower level of recursion with the success bit reset, control is transferred directly to the macro string beginning at location FAIL.

(ii) The PROD macro.

The PROD macro has the form

PROD DEF, FAIL, BACK, M, N

PROD causes the recursive execution (in exactly the same way as RECR) of the macro string beginning at DEF up to N times,

or until a non-successful return is encountered. The exit procedure is as follows:

(a) If the number of successful returns from DEF is at least M, but not more than N, and BACK = 0, continue directly in the current macro string.

(b) In the same situation if BACK = 1, set the success bit equal to 1 and transfer control to the next higher level of recursion.

(c) If the number of successful returns is less than M or more than N and FAIL is not zero, transfer control directly to location FAIL.

(d) In the same situation, if FAIL equals zero, reset the success bit to 0 and transfer control the next higher level of recursion.

(iii) The EXAM Macro.

The EXAM macro has the form

EXAM RØUT, FAL, BACK, PAR

EXAM causes the execution (as an ordinary, and often FORTRAN written procedure) of the subroutine ROUT, supplying it with the parameter PAR; the routine returns in the normal way and either sets or resets the success bit. Thus, for example, the routine may be one that compares the current item of the input string with the parameter supplied it and sets or resets the success bit on finding equality or inequality.

As noted above, certain particular subroutines called by

EXAM, in particular RFIND, may access the master hash table (VARLIST) and may place a recognized element on top of the argument pushdown stack (PDL).

The actions on return from a subroutine called by EXAM are the same as for the RECR and PROD macros.

(iv) The GENR macro.

The GENR macro has the form

GENR RØUT. TRNS, BACK, PAR

GENR causes the execution of the generator subroutine ROUT supplying it with the parameter PAR. A generator routine called in this way will typically generate a porition of the output string (the object language or some related intermediate language) and/or update internal tables (e.g. VARLIST or POL) using the parameter and the internal tables as data.

On return from a called subroutine ROUT, GENR will:

(a) return control to the next higher level of recursion if the BACK bit is set;

(b) transfer control directly to the label TRNS if BACK = 0 and TRNS is not equal to zero;

(c) continue in the current string if TRNS equals zero.

Two additional utility macros SET and TEST are also provided. The SET macro has the form

SET SW, BIT

SET sets a one bit switch, switch number SW, to the value

of BIT. It aas found in practice that providing a small number of these switches (less than 100) is quite useful. The TEST macro has the form

```
TEST    SW, BIT, BACK, FAL
```

This tests the numbered switch SW for equality with BIT and sets or resets the success bit accordingly.

The actions then taken are the same as described for RECR above.

4. An example of the compiler macro coding

The following extracted form the compiler and accomplishes the translation of IF statements; labels used in the extract below generally correspond to those used in the BNF syntax description for the IF statement given in Section 2 of this Appendix. The reader is advised to follow the macro-code below using the material in Section 2 as a guide.

```
*
 IF        EXAM      RCOMPAR,LP,MISCD,0
           SET   2,0
           RECR IFTAIL,ERRIF,0
           GENR    RIF,0,STRT,0
 IFTAIL    RECR   LOGEXP,IFTAILB,0
           EXAM RCOMPAR,RP,ERRRP,0
           TEST    2,1,THRLAB,0
           RECR STORBR,ERR,1
 IFTAILB   RECR MISC,ERR,1
*
*                                  LOGICAL ********** LOGICAL ******* LOGICAL
*
 LOGEXP    RECR CONJ,ERR,0
           PROD     LOGEXP1,ERROR,1,0,77778
 LOGEXP1   EXAM RCOMPAR,OR,0,0
           SET    2,1
           RECR CONJ,ERR,0
           GENR     ARITH,LOR,0,1
 CONJ      RECR LOGTRM,ERR,0
           PROD     CONJ1,0,1,0,77778
 CONJ1     EXAM RCOMPAR,AND,0,0
           SET    2,1
           RECR     LOGTRM,ERR,0
           GENR     ARITH,LAND,0,1
 LOGTRM    EXAM    RCOMPAR,NOT,LOGTR1,0
           SET    2,1
           RECR     LOGELM,ERROR,0
           GENR     ARITH,LNOT,0,1
 LOGTR1    RECR     LOGELM,ERROR,1
 LOGELM    EXAM RCOMPAR,LP,LOGELMA,0
           RECR LOGEXP,ERR,0
           EXAM RCOMPAR,RP,ERR,1
*
*                                  RELATIONS ******* RELATIONS ****** RELATIONS
*
 LOGELMA   RECR RELAT,ERR,1
 RELAT     RECR EXPR,ERR,0
           RECR RELOP,SUC,0
           SET    2,1
           RECR     EXPR,ERR,0
           GENR    RARITH,0,0,1     ,RARITH SETS UP PAR FOR RELATION
 RELOP     EXAM   RCMPAR1,RELIST,0,1
 STORBR    RECR TWOLAB,STORBRA,1
 TWOLAB    EXAM   RFIND,STATNO,0,0,0
           EXAM RCOMPAR,COMMA,ERR,0
           EXAM   RFIND,STATNO,ERR,1
 THRLAB    RECR   TWOLAB,ERR,0
           EXAM RCOMPAR,COMMA,ERR,0
           EXAM   RFIND,STATNO,ERR,1
 STORBRA   GENR    RIF,0,0,0
           RECR     DIST1,0,1
 MISC      EXAM      RCOMPAR,ENDF,ERRIF,0
           RECR MSCTL,ERR,1
 MISCD     EXAM        RCOMPAR,ACCUM,MISCE,0
           RECR CMSTL,ERR,1
 MISCE     EXAM       RCOMPAR,QUO,ERRLP,0
           RECR CMSTL,ERR,1
```

```
MSCTL    EXAM RFIND,INTEGER,ERR,0
         EXAM RCOMPAR,RP,ERR,0
         GENR       RENF,0,0,0
         RECR TWOLAB,ERR,1
CMSTL    GENR       ROVFL,0,0,0
         RECR TWOLAB,ERR,1
SUBSCR   RECR EXPR,ERR,0
         PROD CEXPR,0,1,0,2
CEXPR    EXAM RCOMPAR,COMMA,0,0
         RECR       EXPR,ERR,1
*
*                              EXPR ********** EXPR ********** EXPR ***
*
EXPR     EXAM RCOMPAR,PLUS,EXPR2,0
EXPR1    RECR       TREXP,ERR,1
EXPR2    EXAM RCOMPAR,MINUS,EXPR1,0
         RECR TREXP,ERR,0
         GENR       ARITH,UMIN,0,1
TREXP    RECR TERM,0,0
         PROD SGNTRM,ERR,1,0,8
SGNTRM   EXAM RCOMPAR,PLUS,SGNA,0
         RECR TERM,ERR,0
         GENR     ARITH,PLS,0,1
SGNA     EXAM RCOMPAR,MINUS,0,0
         RECR TERM,ERR,0
         GENR     ARITH,MIN,0,1
TERM     RECR FACTOR,ERR,0
         PROD OFACTOR,0,1,0,8
OFACTOR  EXAM RCOMPAR,TIMES,OFCTA,0
         RECR FACTOR,0,0
         GENR     ARITH,TIM,0,1
OFCTA    EXAM RCOMPAR,DIVIDED,0,0
         RECR FACTOR,0,0
         GENR ARITH,DIV,0,1
FACTOR   RECR BEXP,ERR,0
         EXAM RCOMPAR,EXPONEN,FACA,0
         RECR BEXP,ERR,0
         GENR ARITH,POWER,0,1
FACA     GENR     SUC,0,0,1
         RECR     EXPR,ERR,0
         EXAM RCOMPAR,RP,ERR,1
BXA      RECR       ARRY,BXB,1
BXB      EXAM      RFIND,SIMPLE,BXC,0
         EXAM       RCOMPAR,LP,SUC,0
         RECR     PAREXP,0,0,0
         EXAM      RCOMPAR,RP,0,1
*
*                              PAREXP *********  PAREXP ********* PAREXP *
*
PAREXP   RECR EXPR,ERR,0
        PROD CEXPR,0,1,0,8
BXC      EXAM       RFIND,CNST,ERR,1
ARRY     EXAM       RFIND,ARRAYNM,0,0
         EXAM       RCOMPAR,LP,BXD,0
         RECR        SUBSCR,ERR,0
         GENR RSUBSC,0,0,0
         EXAM        RCOMPAR,RP,ERR,1
BXD      GENR        RVAR,0,SUC,0
```

```
*
*                              FUNCTION ******** FUNCTION ******* FUNCTION
*
 FNST      EXAM       RFIND,SIMPLE,ERR,0
           EXAM         RCOMPAR,LP,AS1,0
           RECR  PAREXP,ERR,0
           EXAM       RCOMPAR,RP,ERR,0
           GENR       FUNCS,0,0,1
 FUNCTX    GENR  FUNC,0,0,0
           RECR  PAREXP,ERR,0
           EXAM  RCOMPAR,RP,ERR,1
*
 BEXP      EXAM  RCOMPAR,LP,BXA,0
           RECR       LOGEXP,0,0,0
           EXAM        RCOMPAR,RP,0,1
```

5. Future Instructions.

It is expected that a later version of our parallel compiler will include the following instructions:

(i) DATA LØCK A

This forces a wait on all accesses to the variable or array A, until execution of a DATA UNLOCK statement.

(ii) DATA RELEASE A

This returns any CPU that accesses the variable or array A to the executive system.

(iii) DATA JUMP A TØ S

This causes any CPU that accesses the variable or array A to jump to location S.

(iv) DATA UNLØCK A

This causes the resetting of any of the above interlocks on the variable A.

These instructions enable the handling of intermediate storage in a particularly simple manner; they also provide means for CPU intercommunication which is quite efficient in many cases.

APPENDIX II

USERS SUMMARY OF THE FORTRAN HARDWARE COMPILER-SIMULATOR

1. Describing a Subassembly

(a) A subassembly is described by a Fortran subroutine, having N + 1 + E arguments. Here:

(ai) N is the number of external lines, or external bundles of lines, leading into or out of the subassembly, from or to the simulated hardware.

(aii) One argument is reserved for the symbolic name of the subassembly, which is a left-justified Hollerith constant of up to 7 characters. This Hollerith constant will appear on the output listing as part of the 'nested context' name associated with each gate and line of the total component listing.

(aiii) The third group of E arguments, which are optional, may be used to convey any additional integers or other information of significance for the given submodule.

EXAMPLE: SUBRØUTINE MØDULE (IN1,IN2,IØUT,NAM,NUMINS1,NUMINS2) for a module with two bundles of inputs, and one output, where the numbers of inputs may be specified.

(b) The first executable statement in the subroutine must be

CALL NAME(NAM)

where NAM is the subroutine parameter in which the module symbolic name will be delivered. This call identifies the module symbolically to the hardware compiler.

(bi) The next group of executable statements in the subroutine must have the form

CALL NAMEL(LINE, left-justified Hollerith line name)

One such statement must appear for each variable representing a line of the submodule which is not an argument of the subroutine. These statements identify the FORTRAN variable LINE as representing a wire of the submodule, and assign it a symbolic name.

EXAMPLES: CALL NAMEL(LINØUT,6LØUTPUT)
CALL NAMEL(L1,6LIJANDK)

(c) If groups (i.e. arrays, i.e. bundles) of lines are involved, the utility function NAMEGEN provided as part of the simulator system may be used to generate symbolic names for lines. This has the form NAMEGEN (Hollerith constant). The hollerith constant must be left justified, 3 characters. NAMEGEN will then prefix successive BCD integers to generate unique names.

EXAMPLE: DØ 1 J=1,7
1 CALL NAMEL(LINES(J),NAMEGEN(3LGAT))

has the same effect as

CALL NAMEL(LINES(1), 7LGAT0000)
CALL NAMEL(LINES(2), 7LGAT0001)
CALL NAMEL(LINES(3), 7LGAT0002)
CALL NAMEL(LINES(4), ... etc.

(d) The next group of executable statements must be a succession of calls on subroutines describing the various submodules of the module described by the total subroutine. These submodules may either be composite submodules described by other user-written subroutines or may be one of the

following elementary submodules provided as part of the simulator system:

```
N1,N2,N3,N4          -- nand gates, 1 to 4 inputs
   D2,D3,D4          -- dot-ands, 2 to 4 inputs
FF11,FF12,FF13,FF14,FF21,FF22,FF32,FF42,
FF31,FF32,FF33,FF34,FF41,FF42,FF43,FF44
                     -- flipflops of 1 to 4 set and
                        reset inputs.
```

These submodules have the evident number of input lines; nand gates and dots have 1 output line, while flipflops have a direct and a complement output line.

EXAMPLES:

```
CALL FF12(ISET,IRESET,JRESET,IØUT,IØUTC,7LCTRLBIT,
          IPHASE)
CALL N2(I1,I2,JØUT,5LGATIN)
CALL FULLADD(I,J,K,LØWBIT,IHBIT,5LADDR8)
```

(e) The final group of executable statements should be

```
CALL UNNAME
RETURN
```

The CALL UNNAME statement keeps the system's internal name context accounting in current status.

2. The Main Program for Compiling and Exercising a Total Assembly.

A full assembly to be compiled and exercised will itself be described by a (highest level) subroutine of the type described above. This subroutine should then be called from a main routine as described in the following paragraph. A call in appropriate form will cause the static compilation

and dynamic exercise of the full assembly; applicable diagnostics will be printed, etc.

(a) The first executable statement of the main program should be

(label) CALL STEP(N)

where N is the desired number of simulation steps to be carried out. This call initializes all system tables necessary for the subsequent compilation-simulation. If N = 0, a hardware compilation will be performed but no simulation attempted.

EXAMPLES: CALL STEP(0)
CALL STEP(150)

(bi) The next group of executable statements should be a collection of calls

CALL NAMEL(LINE, left-justified hollerith line name) .

One such statement must occur for each variable representing an external output line of the assembly being compiled, that is, a line which is an output from some gate, flipflop, dot, or other subassembly of the full assembly. These statements identify the FORTRAN variable LINE as representing a wire of the assembly, and assign it a symbolic name. (Cf. 1.ci. above, and the examples given there.)

(bii) Input wires to the subassembly, that is, wires providing signals to the assembly from hypothetical external equipment, but which are not outputs of any gate, flipflop, dot, or other subassembly, should be specified to the system by the alternate call

CALL INAMEL(LINE, left justified Hollerith line name, N)
where the additional parameter N specifies the number of times the designated line may be used as an input without overloading.

EXAMPLES: CALL INAMEL(IN1, 6LINPUTA,8)
CALL INAMEL(L11, 7L66CHAN1,24)

If a group (i.e. an array or bundle) of input lines is to be described, the utility function NAMEGEN may be used to generate symbolic names as arguments to the subroutine INAMEL; thus

EXAMPLE:
```
      DØ 1 J=1,7
    1 CALL INAMEL(LINES(J),NAMGEN(3LGAT),8)
```
is valid. (Cf. 1.ci. above for an additional discussion of NAMGEN.)

(c) The next group of executable statements is required only if simulation of the subassembly is intended. This group of statements consists of a set of subroutine calls of the form

CALL SET(LINE,LINVAL)

which sets the value of the input line LINE to one of the boolean LINVALues 0 or 1. Any input line to the assembly not SET in this manner will have an undefined boolean value during subsequent simulation.

EXAMPLE: CALL SET(INLIN,1)

A separate value of the input line value may be specified for every subsequent step of simulation. As a convenient source of input values a utility function KØUNT is provided.

This has the form KØUNT(N,M), whose value on the K-th call to KØUNT is the N-th boolean bit of the integer K/M.

```
EXAMPLE:    CALL SET(IN1, KØUNT(1,3))
            CALL SET(IN2, KØUNT(2,3))
            CALL SET(IN3, KØUNT(3,3))
```

(d) Next should follow a call to the 'main' subroutine defining the assembly to be compiled and exercised.

```
EXAMPLE:    CALL ADDER(IN1,IN2,IN3,LØWBIT,IHBIT,5LADDER)
```

(e) The next group of executable statements is required only if simulation of the subassembly, and printout of the results of the simulation, is desired. This group consists of a set of statements

```
            J(K) = LINE
```

defining the successive locations of a dimensioned array with the various lines whose values are to be printed during simulation, and of a single additional statement

```
            CALL PNTL(N,J)
```

to the simulation-print routine PNTL. The first argument N of PNTL is the number of lines represented in the dimensioned array J.

(f) The final executable statement of the main program should be the transfer

```
            GØ TØ label
```

which returns to the first executable statement (which, cf. (a) above, has the form CALL STEP(N)).

3. Flipflop Phases and Clock Signals.

The final IPHASE argument to the built-in flipflop routines FF11,FF12,..., etc. (Cf. 1.d. above for details) is a set of bits (up to 9 in number) describing the clock phases on which the flipflop described by a call to one of these subroutines is allowed to vary. If, during simulation, a flipflop is found to vary on an illegal clock phase, a diagnostic statement will be printed, but simulation will be continued. (The simulator is presently provided with a simulated two-phase internal clock, but the number of phases may trivially be increased to a maximum of 9.) If a flipflop is to change only on clock phases 1,3,5,...,etc., it should be called with its final argument IPHASE=1; if it is to change only on clock phases 2,4,6,...,etc., it should be called with its final argument IPHASE=2. A flipflop which may change either on even or odd phases should have both phase bits set, and should consequently be called with its final argument IPHASE=3 (since 3 = 1.ØR.2).

The clock phase bits may be used as gating signals via the built in function KLØK. KLØK(1) has the value 1 on the odd clock phases, KLØK(2) has the value 1 on even clock phases.

The statement

```
CALL SET(J,KLØK(N))
```

will put the simulated clock signal (even or odd phases, depending on N) on the line J; in this way, appropriate clocking

of the various submodules of the simulated hardware can be achieved.

APPENDIX

SOME SAMPLE PROGRAMS

```
C  **** SAMPLE EXCLUSIVE ØR SUBRØUTINE ***************
C        ***** ***** *****   SAMPLE ***** ***** *****
       SUBRØUTINE NØR(I,J,K,NAM)
C      SAMPLE SUBRØUTINE TØ EXEMPLIFY GENERAL SCHEME
C      BUILD UP ADDITIØNAL CØNTEXT
       CALL NAME(NAM)
C      ASSIGN NAMES TØ INTERNAL LINES
       CALL NAMEL(KKK,3LKKK)  $  CALL NAMEL(LLL,3LLLL)
       CALL NAMEL(LL,2LLL)
C      DESCRIBE 4 FLIPFLØPS, ASSIGN GATE NAMES
       CALL N2(I,J,LL,5LGATIN)
       CALL N2(I,LL,LLL,4LGAT+)
       CALL N2(J,LL,LLL,4LGAT-)
       CALL N2(LLL,KKK,K,5LGATØU)
C      REMØVE INSERTED CØNTEXT AND RETURN
       CALL UNNAME
       RETURN
       END

C  ****  SAMPLE PRØGRAM SHØWING THREE BIT TØ TWØ BIT FULL ADDER
C         ***** ***** *****   SAMPLE ***** ***** *****
       PRØGRAM HWTEXT(INPUT=100,ØUTPUT=100,TAPE1=ØUTPUT,TAPE2=INPUT)
       DIMENSIØN IPNT(5)
  1    CALL STEP(17)
       CALL INAMEL (I,1LI,8)$  CALL INAMEL(J,1LJ,8)
       CALL INAMEL(K,1LK,8)
       CALL NAMEL(IJ,2LIJ)
       CALL NAMEL(JI,2LJI)
       IPNT(1)=I $ IPNT(2)=J  $ IPNT(3)=K $ IPNT(4)=IJ $ IPNT(5)=JI
       CALL SET(I,KØUNT(1,3))$CALL SET(J,KØUNT(2,3))$CALL SET(K,
      XKØUNT(3,3))
```

[continued]

```
      CALL ADDR(I,J,K,LJ,JI,7LADDUNIT)
      CALL PVAL(5,IPNT)
      GØ TØ 1
      END
      SUBRØUTINE ADDR(I,J,K,II,JJ,NAM)
      CALL NAME(NAM)
      CALL NAMEL(NIAJ,4LNIAJ) $  CALL NAMEL(NIAK,4LNIAK)
      CALL NAMEL(NJAK,4LNJAK) $  CALL NAMEL(NIJK,4LNIJK)
      CALL NAMEL(INJK,4LINJK) $  CALL NAMEL(JNIK,4LJNIK)
      CALL NAMEL(KNIJ,4LKNIJ)
      CALL N2(I,J,NIAJ,1LA) $  CALL N2(I,K,NIAK,1LB)
      CALL N2(J,K,NJAK,1LC) $  CALL N3(NIAJ,NIAK,NJAK,II,1LD)
      CALL N3(I,J,K,NIJK,1LE) $CALL N3(I,NIAJ,NIAK,INJK,1LF)
      CALL N3(J,NJAK,NIAJ,JNIK,1LG) $ CALL N3(K,NJAK,NIAK,KNIJ,1LH)
      CALL N4(INJK,JNIK,KNIJ,NIJK,JJ,1LØ)
      CALL UNNAME
      RETURN
      END
C  ***** SAMPLE PRØGRAM SHØWING PARITY TREE WRITTEN IN TERMS ØF
C       EXCLUSIVE ØR *** ***** SAMPLE ***** ***** *****
      PRØGRAM HWTEXT(INPUT=100,ØUTPUT=100,TAPE1=ØUTPUT,TAPE2=INPUT)
      DIMENSIØN IPNT(5)
      CALL STEP(17)
      CALL INAMEL(I,1LI,8)  $  CALL INAMEL(J,1LJ,8)
      CALL INAMEL(K,1LK,8)  $  CALL INAMEL(L,1LL,8)
      CALL NAMEL(IJKL,4LLINØ)
      IPNT(1)=I  $  IPNT(2)=J  $  IPNT(3) = K
      IPNT(4)=L  $  IPNT(5)=IJKL
      CALL SET(I,KØUNT(1,4)) $  CALL SET(J,KØUNT(2,4))
      CALL SET(K,KØUNT(3,4)) $  CALL SET(L,KØUNT(4,4))
      CALL TREE4(I,J,K,L,IJKL,5LTREE4)
      CALL PVAL(5,IPNT)
      GØ TØ 1
      END
```

[continued]

```
      SUBRØUTINE TREE4(I,J,K,L,IJKL,NAM)
C     SUBRØUTINE CALLING DAND
      CALL NAME(NAM)
      CALL NAMEL(IJ,2LIJ)  $  CALL NAMEL(KL,2LKL)
      CALL DAND(I,J,IJ,3LØR+)  $  CALL DAND(K,L,KL,3LØR-)
      CALL DAND(IJ,KL,IJKL,5LØRØUT)
      CALL UNNAME
      RETURN
      END
      SUBRØUTINE DAND(I,J,IJ,NAM)
C     SUBRØUTINE USING AND-DØT FØR EXCLUSIVE ØR
      CALL NAME(NAM)
      CALL NAMEL(NI,2LNI)  $  CALL NAMEL(NJ,2LNJ)
      CALL NAMEL(IAJ,3LIAJ)  $  CALL NAMEL(NIJ,3LNIJ)
      CALL N1(I,NI,4LINVI)  $  CALL N1(J,NJ,4LINVJ)
      CALL N2(I,J,IAJ,2LG1)  $  CALL N2(NJ,NI,NIJ,2LG2)
      CALL D2(IAJ,NIJ,IJ,3LDØT)
      CALL UNNAME
      RETURN
      END
C ****  SAMPLE PRØGRAM SHØWING TWØ BIT SHIFT REGISTER ***** *****
       ***** ***** *****    SAMPLE ***** ***** *****
      PRØGRAM HWTEXT(INPUT=100,ØUTPUT=100,TAPE1=ØUTPUT,TAPE2=INPUT)
      DIMENSIØN IPNT(5)
 1    CALL STEP(17)
      CALL INAMEL(I,1LI,8) $  CALL INAMEL(J,1LJ,8)
      CALL INAMEL(K,1LK,8)
      CALL NAMEL(IJ,2LIJ)
      CALL NAMEL(JI,2LJI)
      CALL SET(I,1) $CALL SET(J,KØUNT(1,2))$ CALL SET(K,KØUNT(2,2))
      IPNT(1)=I $ IPNT(2)=J $ IPNT(3)=K $ IPNT(4)=IJ $ IPNT(5)=JI
      CALL SHIF2(I,J,K,IJ,JI,5LSHIF2)
      CALL PVAL(5,IPNT)
      GØ TØ 1
      END
      SUBRØUTINE SHIF2(I,J,K,LL1,MM1,NAM)
C     2-BIT CIRCULAR SHIFT FØR TESTING
      CALL NAME(NAM)
      CALL NAMEL(L1,2LLI)  $  CALL NAMEL(L2,2LL2)
      CALL NAMEL(M1,2LM1)  $  CALL NAMEL(M2,2LM2)
           CALL NAMEL(LL2,3LLL2)
           CALL NAMEL(MM2,3LMM2)
      CALL N2(I,LL1,L1,4LGLL1) $ CALL N2(I,LL2,L2,4LGLL2)
      CALL N1(J,M1,4LGMM1)  $  CALL N2(K,MM2,M2,4LGMM2)
      CALL FF11(L1,L2,MM1,MM2,4LEVEN,1) $ CALL FF11(M1,M2,LL1,
     XLL2,3LØDD,2)
      CALL UNNAME
      RETURN
      END
                          [end]
```

APPENDIX III

S O A P S U D S

USER'S MANUAL

SECOND EDITION

FOR VERSION 2.1

MARCH 10, 1967

RALPH GRISHMAN

ACKNOWLEDGEMENTS

THE ORIGINAL DESIGN FOR THE ATHENE MULTIPROCESSOR SYSTEM WHICH IS SIMULATED BY SOAPSUDS WAS DEVELOPED BY J. SCHWARTZ IN EARLY 1965.

A CONSIDERABLE PORTION OF THE CODE OF SOAPSUDS IS AN ADAPTATION OF THE SIMULATION ROUTINES IN WATCHR, A DEBUGGING SYSTEM FOR THE CDC6600 WRITTEN IN EARLY 1965 BY J. SCHWARTZ AND J. GLASNER. IN THE DEBUGGING OF SOAPSUDS, AND IN THE DESIGN OF ITS VARIOUS OPTIONS, SINCE THE SUMMER OF 1965, I HAVE RECEIVED MUCH ASSISTANCE FROM E. DRAUGHON, WHO WAS CONCURRENTLY DEBUGGING AND EXPANDING THE WATCHR SYSTEM. HE HAS ALSO DESIGNED ONE OF THE PROGRAMMING SYSTEMS TO BE USED WITH SOAPSUDS (DESCRIBED IN APPENDIX II). THANKS ARE ALSO DUE TO AARON STEIN, WHO HAS WRITTEN A NUMBER OF PROGRAMS TO TEST THE SOAPSUDS SYSTEM, AND HAS OFFERED ME MUCH ASSISTANCE IN THE WRITING OF THIS MANUAL.

R. GRISHMAN

INTRODUCTION

SOAPSUDS IS SCHWARTZ'S OWN ATHENE PROCESSOR, SERIAL UNIPROCESSOR DEBUGGING SIMULATOR. IT IS A PROGRAM CODED FOR THE CDC 6600 AND DESIGNED TO SIMULATE A SET OF SIXTY CENTRAL PROCESSING UNITS, SIMILAR TO THE CDC 6600, OPERATING IN PARALLEL FROM A COMMON MEMORY. IN ADDITION TO THE INSTRUCTION REPERTOIRE OF THE CDC 6600, A NUMBER OF INSTRUCTIONS SUITABLE FOR A MULTI-PROCESSOR ENVIORNMENT HAVE BEEN IMPLEMENTED.

THE SIMULATOR HAS BEEN DESIGNED FOR DEBUGGING AND ESTIMATING THE EFFICIENCY OF MULTI-PROCESSOR PROGRAMS, AND FOR STUDYING THE CHARACTERISTICS OF LARGE SCALE MULTI-PROCESSOR SYSTEMS, IN ORDER TO OBTAIN QUANTITATIVE DATA FOR THE DESIGN OF FUTURE SYSTEMS OF THIS TYPE. TO THIS END, A NUMBER OF OPTIONS ARE AVAILABLE, FOR TRAPPING AND TRACING INSTRUCTIONS, CHECKING THE VALUES OF SPECIFIED LOCATIONS, AND MEASURING THE RUNNING TIMES OF ROUTINES.

THE ATHENE SYSTEM

THE SOAPSUDS PROGRAM IS DESIGNED TO SIMULATE A CONFIGURATION OF CENTRAL PROCESSING UNITS KNOWN AS ATHENE 1. THIS PROCESSING SYSTEM WAS DESIGNED BY PROF. JACK SCHWARTZ, WHO COINED THE TERM ATHENE PROCESSOR (1).

THE TERM ATHENE PROCESSOR REFERS TO A SET OF CENTRAL PROCESSING UNITS, EACH WITH ITS OWN INSTRUCTION LOCATION COUNTER, OPERATING FROM A COMMON MEMORY. SOAPSUDS PROVIDES FOR THE SIMULATION OF UP TO SIXTY PROCESSORS.

EACH PROCESSOR IS SIMILAR TO THE CENTRAL PROCESSING UNIT OF THE CDC 6600. IT HAS 24 ARITHMETIC REGISTERS (THE A, B, AND X REGISTERS), AND ALL THE INSTRUCTIONS WHICH ARE ON THE 6600, USING THE SAME OPCODES AS THE 6600. EACH PROCESSOR HAS ITS OWN INSTRUCTION LOCATION COUNTER AND UPPER AND LOWER MEMORY BOUNDS (A PROCESSOR CANNOT LOAD, STORE, OR EXECUTE CODE OUTSIDE ITS OWN MEMORY BOUNDARIES). IN ADDITION, EACH PROCESSOR HAS A SPECIAL REGISTER, KNOWN AS THE ASSIGNED BADGENUMBER REGISTER, WHICH MAY BE READ BY THE PROCESSOR WITH ONE OF THE EXTRA INSTRUCTIONS, READ BADGENUMBER. THE BADGENUMBER REGISTER CAN BE SET ONLY BY A SPECIAL SUPERVISORY COMPUTER, WHOSE SIMULATION HAS NOT BEEN IMPLEMENTED ON THE PRESENT SOAPSUDS SYSTEM.

THE INSTRUCTIONS WHICH HAVE BEEN ADDED TO THE 6600 REPERTOIRE ARE

1. READ BADGENUMBER (RBN BI)

```
-------------------------
- 0 0 1 0 I 0 0 0 0 0 -
-------------------------
```

THIS INSTRUCTION PLACES THE CONTENTS OF THE ASSIGNED BADGENUMBER REGISTER IN B REGISTER NUMBER I. IF I=0, THE INSTRUCTION IS A NO-OP.
THE OPERATION CODE IS 001, AND THE INSTRUCTION IS 30 BITS LONG. THE REGISTER NUMBER APPEARS IN BITS 15-17 (LOW ORDER BIT = BIT 0).

2. REPLACE ADD (RAD XI,BJ+K)

```
-------------------------
- 0 0 I J A A A A A A -
-------------------------
```

THIS INSTRUCTION STORES THE 60-BIT ONE'S-COMPLEMENT SUM OF REGISTER XI AND THE CONTENTS OF LOCATION BJ+K BOTH IN REGISTER XI AND AT LOCATION BJ+K. IF TWO OR MORE PROCESSORS SIMULTANEOUSLY REACH A REPLACE ADD INSTRUCTION REFERENCING THE SAME LOCATION, THE PROCESSORS WILL EXECUTE THE INSTRUCTION SERIALLY (IN ORDER BY PROCESSOR NUMBER). REPLACE ADD INSTRUCTIONS CAN ONLY BE PERFORMED WITH REGISTERS X6 AND X7, SO THE OPERATION CODES ARE 006 (REPLACE ADD X6) AND 007 (REPLACE ADD X7). THE INSTRUCTION IS 30 BITS LONG. THE ADDRESS PORTION K (AAAAAA) IS IN BITS 0-17, AND THE NUMBER OF THE B REGISTER (J) ADDED TO K TO COMPUTE THE EFFECTIVE ADDRESS IS GIVEN IN BITS 18-20

3. EXIT (XIT BJ+K)

```
-------------------------
- 0 0 5 J A A A A A A -
-------------------------
```

IN ADDITION TO THE REGISTERS DESCRIBED ABOVE, EACH CPU HAS A TOGGLE KNOWN AS THE PROGRAM/RESIDENT MODE INDICATOR. THIS INDICATOR AFFECTS ONLY THE EXECUTION OF ONE INSTRUCTION, THE EXIT INSTRUCTION, TO BE DESCRIBED BELOW.

ALL THE PROCESSORS SHARE IN COMMON A SET OF 3 ADDRESSES, THE RESIDENT UPPER LIMIT, THE RESIDENT LOWER LIMIT, AND THE RESIDENT EXIT ADDRESS. WHENEVER THE PROGRAM/RESIDENT MODE INDICATOR IS IN PROGRAM MODE, AND AN EXIT INSTRUCTION IS EXECUTED, THE UPPER AND LOWER MEMORY LIMITS FOR THE PROCESSOR ARE SET TO THE RESIDENT MEMORY LIMITS, THE PROCESSOR TRANSFERS CONTROL TO THE RESIDENT EXIT ADDRESS, AND THE INDICATOR CHANGES TO PROGRAM MODE. ON THE OTHER HAND, IN RESIDENT MODE, THE CONTENTS OF LOCATION BJ+K IS USED TO DETERMINE THE NEW UPPER AND LOWER MEMORY LIMITS AND P REGISTER. THE HIGH 20 BITS (40-59) SPECIFY THE ADDRESS TO WHICH CONTROL IS TRANSFERRED, AND THE MIDDLE 20 BITS (20-39) AND LOW 20 BITS (0-19) BECOME THE LOWER AND UPPER MEMORY LIMITS, RESPECTIVELY. THE INDICATOR IS THEN SET TO PROGRAM MODE. THUS REPEATED EXECUTION TOGGLES THE PROCESSOR BETWEEN RESIDENT AND PROGRAM MODES.

THIS INSTRUCTION WAS DESIGNED TO FACILITATE THE IMPLEMENTATION OF AN OPERATING SYSTEM UNDER SOAPSUDS. USING THIS INSTRUCTION, MEMORY LIMITS CAN BE ALTERED AS NEEDED, SO THAT ONE PROGRAM IN THE SYSTEM WILL BE PREVENTED FROM WRITING OVER ANY OTHER PROGRAM. FURTHERMORE, THE CODE WHICH DETERMINES THE NEW LIMITS IS IN A SPECIALLY PROTECTED AREA OF MEMORY, SO NO PROGRAM MAY ACCIDENTALLY ALTER ITS MEMORY LIMITS TO INCLUDE ANOTHER PROGRAM.

A MAJOR SECTION OF SOAPSUDS IS AN OFFSHOOT OF WATCHER, A DEBUGGING AID FOR THE 6600 WHICH SIMULATES THE 6600 CENTRAL PROCESSOR(2). SOAPSUDS, LIKE WATCHER, USES THE PROGRAM TO BE SIMULATED AS DATA, ANALYZING THE INSTRUCTIONS AND PERFORMING THE OPERATIONS THEY REQUEST. A BLOCK IN MEMORY IS RESERVED TO HOLD THE SIMULATED REGISTERS FOR EACH PROCESSOR.

IT TAKES ROUGHLY 80 TIMES AS LONG FOR SOAPSUDS TO SIMULATE AN INSTRUCTION AS IT DOES FOR THE 6600 TO ACTUALLY EXECUTE AN INSTRUCTION. HENCE, IF ALL 60 SIMULATED PROCESSORS ARE RUNNING, IT TAKES ABOUT 60*80 TIMES AS LONG FOR SOAPSUDS TO SIMULATE ONE INSTRUCTION FOR ALL THE PROCESSORS AS IT DOES FOR THE 6600 TO EXECUTE ONE INSTRUCTION. THIS IMPLIES A MAXIMUM SPEED ON THE ORDER OF ONE INSTRUCTION PER MILLISECOND, 1000 INSTRUCTIONS PER SECOND PER COMPUTER.

INCLUDED IN SOAPSUDS IS A TRAPPING SYSTEM WHEREBY, WHEN A SPECIFIED OCCURANCE TAKES PLACE SOAPSUDS TEMPORARILY STOPS SIMULATION AND TRANSFERS CONTROL TO A TRAP ROUTINE. TRAPS MAY OCCUR ON THE EXECUTION OF A PARTICULAR OPCODE BY A PARTICULAR PROCESSOR, ON TRANSFER OF CONTROL TO A PARTICULAR LOCATION, OR ON A LOAD FROM OR STORE TO A PARTICULAR LOCATION. NOTE THAT CONTROL IS ACTUALLY TRANSFERRED TO THE TRAP ROUTINE, I.E., THE ROUTINE IS EXECUTED DIRECTLY BY THE 6600, NOT SIMULATED BY SOAPSUDS.

ALONG WITH PROVISIONS FOR TRAPPING, SOAPSUDS HAS A COMPLETE SET OF TRACES. ANY EVENT WHICH MAY BE TRAPPED ON MAY ALSO BE TRACED. IN SOAPSUDS, TRACING IS SIMPLY A SPECIAL TRAP, WHICH TRAPS TO A ROUTINE THAT PRINTS A MESSAGE (TRACE). ROUTINES IN SOAPSUDS TURN ON AND OFF TRAPS AND TRACES, AND SUSPEND THEM TEMPORARILY.

IN ADDITION TO TRAPPING AND TRACING, THERE ARE TWO OPTIONS IN SOAPSUDS FOR CHECKING THE CONTENTS OF SPECIFIED LOCATIONS. ONE OF THE ROUTINES CORRECTS THE CONTENTS OF THE LOCATION IF IT IS INCORRECT, THE OTHER DOES NOT. THESE TWO ROUTINES CHECK THE VALUES OF THE LOCATIONS EACH TIME A STORE IS MADE INTO THE LOCATION.

SOAPSUDS HAS A BUILT IN MEMORY FEATURE WHICH ENABLES THE WRITING OF PARALLEL PROCEDURES IN FORTRAN WITH RELATIVE EASE-- THE ABILITY OF HANDLING LOADS AND STORES TO A USER DEFINED REGION OF MEMORY AS BEING PRIVATE TO THE PROCESSOR THAT EXECUTED THE LOAD OR STORE. I.E., EACH PROCESSOR HAS A UNIQUE LOCATION ASSOCIATED WITH THE ADDRESS OF THE LOAD OR STORE WHICH CONTAINS THE ACTUAL WORD THAT IS BEING LOADED OR STORED. ONE SEES IMMEDIATELY THAT, IF THE ROUTINES THAT ARE TO BE OPERATED IN PARALLEL ARE PLACED IN THE PRIVATE MEMORY REGION, THE COMPILER GENERATED TEMPORARY STORES AND RESTORES AND THE CALLS TO THESE ROUTINES ARE CORRECTLY HANDLED AUTOMATICALLY. FURTHERMORE, DO LOOPS CAN BE EXECUTED COMPLETELY IN PARALLEL WITH EACH CPU HAVING ITS OWN VALUE OF THE DO VARIABLE.
THUS, FOR EXAMPLE, IF NEWVAL IS A FUNCTION THAT INCREMENTS ITS PARAMETER BY 1 IN A SERIAL MANNER (ONLY ONE CPU AT A TIME), J IS DEFINED AS A 'PUBLIC' VARIABLE (I.E., LOADS AND STORES REFERENCING J ARE PERFORMED IN THE USUAL MANNER), JJ IS DEFINED AS A 'PRIVATE' VARIABLE(IN THE REGION OF MEMORY DESIGNATED AS PRIVATE) AND J IS INITIALLY SET TO 0, THE FOLLOWING CODING WILL GIVE EACH CPU EXECUTING IT A DIFFERENT VALUE OF JJ UNTIL THE CPU'S JJ EXCEEDS N, AT WHICH TIME THE CPU IS SHUNTED OFF TO STATEMENT 100.

```
1     JJ = NEWVAL(J)
      IF (JJ .GT. N ) GO TO 100

        (PROGRAM)

      GO TO 1
```

IN PRACTICE, THE VARIABLES AND ARRAYS THAT ARE TO BE PUBLIC (I.E., THOSE VARIABLES THAT ARE REFERENCED BY THE CPU-S IN THE ORDINARY WAY) ARE PLACED IN BLANK OR NUMBERED COMMON, AS THESE ARE ASSIGNED TO THE END OF MEMORY, WHILE PRIVATE VARIABLES OR ARRAYS ARE ASSIGNED IMPLICITY OR BY DIMENSION OR LABLED COMMON STATEMENTS.

DESIGNATING THE MEMORY REGIONS

IN ADDITION TO THE PRIVATE REGION DESCRIBED ABOVE, SOAPSUDS REQUIRES THE SPECIFICATION IF TWO AUXILLIARY REGIONS--

A. THE PRIVATE EXECUTION REGION DESIGNATING THE AREA IN WHICH LOADS AND STORES TO THE PRIVATE REGION ARE HANDLED AS ABOVE (THE COMPLEMENT OF THIS REGION ALLOWS ORDINARY LOADING AND STORING INTO THE PRIVATE AREA FOR SUCH PURPOSES AS RESETTING CODING LINES, ETC.)

B. THE STORAGE REGION IN WHICH SOAPSUDS KEEPS THE PRIVATELY STORED INFORMATION.

THE REGIONS OF MEMORY ARE DEFINED BY A THREE ELEMENT ARRAY WHOSE ADDRESS IS PROVIDED BY THE SIXTH PARAMETER OF THE SOAPSUDS CALL. EACH ELEMENT OF THE ARRAY CONTAINS TWO ADDRESSES.THE LOWER BOUND, OF THE AREA IT DESIGNATES, IN BITS 47-30 AND THE UPPER BOUND IN BITS 17-0. THE FIRST ELEMENT DESIGNATES THE PRIVATE MEMORY BOUNDS, THE SECOND, THE BOUNDS OF THE PRIVATE EXECUTION AREA AND THE THIRD, THE BOUNDS OF THE STORAGE AREA.

SO LONG AS ALL INSTUCTIONS IN THE PROGRAM ARE WITHIN THE SECOND PAIR OF LIMITS ('WITHIN WHICH STORES TO PRIVATE MEMORY ARE TREATED AS PRIVATE STORES'), SOAPSUDS COMPLETELY SIMULATES THE EXISTENCE OF PRIVATE

MEMORY REGIONS, AND THE USER NEED NOT CONCERN HIMSELF WITH THE METHOD OF SIMULATION. HOWEVER, SIMULATED MEMORY ACCESSES OUTSIDE THESE LIMITS WILL NOT PRESERVE THE ILLUSION, SO IT IS POSSIBLE, FOR EXAMPLE, FOR ONE PROCESSOR TO STORE INTO ANOTHER PROCESSOR'S PRIVATE MEMORY. A PROCESSOR OUTSIDE THESE LIMITS, LOADING A WORD IN THE PRIVATE REGION, MUST CHECK WHETHER THE HIGH 42 BITS ARE 00202211260124. IF THEY ARE, THE SIMULATED MEMORY WORDS CORRESPONDING TO THAT ADDRESS ARE CONTAINED IN AN ARRAY WHOSE LOCATION IS GIVEN BY THE LOW 18 BITS. THE LENGTH OF THE ARRAY IS THE NUMBER OF PROCESSORS, AND THE POSITIONS IN THE ARRAY CORRESPOND TO THE PROCESSOR NUMBERS (THE FIRST ELEMENT IS PROCESSOR ZERO'S PRIVATE MEMORY WORD, THE SECOND IS PROCESSOR ONE'S, ETC.). LOADS AND STORES MUST IN GENERAL BE MADE TO THIS ARRAY--A STORE INTO THE PRIVATE REGION WILL IN EFFECT SET THE LOCATION TO THE SAME VALUE FOR ALL PROCESSORS.

USERS WHO MAKE CALLS FROM OUTSIDE THE SECOND LIMITS MUST BE ESPECIALLY CAUTIOUS TO CHECK WHETHER ANY OF THE ARGUMENTS IS A POINTER TO A PRIVATE MEMORY ARRAY (IN WHICH CASE THE ACTUAL ADDRESS IN THE ARRAY MUST BE SUBSTITUTED). FOR CALLS WITHIN THESE LIMITS, SOAPSUDS TAKES CARE OF ALL NECESSARY ARRANGEMENTS.

FOR NORMAL CODING (WITHIN THE SECOND PAIR OF LIMITS) THE USER NEED ONLY MAKE SURE HE ALLOCATES SUFFICIENT SPACE FOR SOAPSUDS TO KEEP THE PRIVATE MEMORY ARRAYS (BETWEEN THE THIRD PAIR OF LIMITS). THE MINIMUM NUMBER OF LOCATIONS IS THE NUMBER OF ADDRESSES IN THE PRIVATE REGION INTO WHICH STORES ARE MADE (VARIABLES+TEMPORARIES+INDIRECTS) TIMES THE NUMBER OF PROCESSORS. IF INSUFFICIENT SPACE IS ALLOTED, SOAPSUDS WILL ERROR EXIT WHEN THE SPACE IS EXCEEDED (SEE ERROR EXITS, BELOW).

USING THE SIMULATOR -- BASIC INFORMATION

SOAPSUDS IS AVAILABLE AS A RELOCATABLE CHIPPEWA BINARY DECK. THE SOAPSUDS PACKAGE CONSISTS OF THE MAIN SOAPSUDS BINARY DECK AND A SET OF CONNECTING SUBROUTINES, IN ASCENTF (SYMBOLIC). THE PROGRAM USING SOAPSUDS NEVER CALLS SOAPSUDS DIRECTLY. INSTEAD, ALL CALLS ARE MADE THROUGH ONE OF THE CONNECTING SUBROUTINES. THE COMPLETE SOAPSUDS PACKAGE MAY BE INCLUDED IN A DECK WITH THE PROGRAM BEING TESTED AS REGULAR BINARY AND SYMBOLIC SUBROUTINES.

ENTRY

SINCE THE USER SUPPLIES THE MAIN PROGRAM, THE PROGRAM IS INITIALLY NOT UNDER SOAPSUDS SIMULATION. TO BEGIN SIMULATION, THE FOLLOWING CALL MUST BE EXECUTED

CALL SOAPSDS(NUMCP,LOLIM,HILIM,START,XTLOC,MMBND)

WHERE NUMCP IS THE TOTAL NUMBER OF PROCESSORS TO BE SIMULATED
LOWLIM IS THE INITIAL LOWER MEMORY LIMIT FOR ALL PROCESSORS, AND THE RESIDENT LOWER MEMORY LIMIT
HILIM IS THE INITIAL UPPER MEMORY LIMIT FOR ALL PROCESSORS, AND THE RESIDENT UPPER MEMORY LIMIT
START IS THE LOCATION AT WHICH ALL PROCESSORS BEGIN EXECUTION
XTLOC IS THE LOCATION TO WHICH CONTROL IS TRANSFERED ON THE EXECUTION OF AN XIT INSTRUCTION BY A PROCESSOR IN PROGRAM MODE (THE RESIDENT EXIT ADDRESS)
MMBND IS THE LOCATION OF THE THREE ELEMENT ARRAY THAT SETS THE PRIVATE MEMORY BOUNDS (SEE DISCUSSION ABOVE)

IF NUMCP IS ZERO, ONE PROCESSOR WILL BE SIMULATED.
THE ASSIGNED BADGENUMBER OF EACH PROCESSOR IS SET TO THE ABSOLUTE BADGENUMBER(I.E., THE NUMBER) OF THE PROCESSOR. THUS PROCESSOR ZERO GETS AN ASSIGNED BADGENUMBER OF ZERO, PROCESSOR ONE AN ASSIGNED BADGE-NUMBER OF ONE, ETC. ALL ARITHMETIC REGISTERS START OUT ZERO.

ANY ATTEMPT TO LOAD, STORE OR TRANSFER CONTROL TO THE SOAPSUDS PROGRAM AREA WILL CAUSE AN OUT OF BOUNDS ERROR. AFTER SOAPSDS HAS BEEN ANY FURTHER CALLS TO SOAPSDS WILL RESULT IN AN ERROR EXIT.

AUTOMATIC CHECKS DURING SIMULATION

THE FOLLOWING EVENTS ARE CHECKED FOR DURING SIMULATION AND ARE CONSIDERED ERROR CONDITIONS

1. PROGRAM STOP (EXIT MODE 0)
2. LOAD OR STORE OUT OF BOUNDS (EXIT MODE 1)
3. TRANSFER OF CONTROL TO A LOCATION OUTSIDE MEMORY BOUNDS (EXIT MODE 2)
4. EXECUTION OF A 30-BIT OPCODE IN THE LOW QUARTER OF A WORD ('EXECUTING DATA') (EXIT MODE 4)
5. INSUFFICIENT SPACE FOR PRIVATE MEMORY ARRAYS (EXIT MODE 8)
6. END IN RA+1 (EXIT MODE 16)
7. ABT IN RA+1 (EXIT MODE 32)
8. ERROR IN A CALL TO A SOAPSUDS OPTION (EXIT MODE 64)

ON ENCOUNTERING ANY OF THESE CONDITIONS, EXCEPT FOR A PROGRAM STOP, SOAPSUDS WILL PRINT OUT A SNAP OF THE PROCESSOR AND A SOAPSNAP, AND, IF THE RESIDENT EXIT ADDRESS (THE FIFTH PARAMETER IN THE CALL TO SOAPSDS) IS ZERO, WILL ABORT. IF IT IS NOT ZERO, AND THE PROCESSOR IS IN RESIDENT MODE, SOAPSUDS WILL ALSO ABORT. HOWEVER, IF IT IS NON-ZERO AND THE PROCESSOR IS IN PROGRAM MODE, SOAPSUDS WILL SET B7 TO 3, WILL CLEAR X0 AND SOTRE THE CURRENT P REGISTER IN BITS 30-47 AND THE EXIT MODE IN BITS 48-59 OF X0, AND WILL XIT TO THE RESIDENT.

IN CASE OF A PROGRAM STOP, WITH A ZERO RESIDENT EXIT ADDRESS, SOAPSUDS WILL SIMPLY STOP THE PROCESSOR, PRINTING NO MESSAGE UNLESS A TRACE OF OPCODE 00 WAS REQUESTED. IF THE RESIDENT EXIT ADDRESS IN NON-ZERO, THE USUAL EXIT PROCEDURE WILL BE FOLLOWED (ALTHOUGH NO SNAP OR SOAPSNAP WILL APPEAR).

CALLING SOAPSUDS OPTIONS--AUTOMATIC TRAPPING

FOR REASONS OF SIMPLICITY, SOAPSUDS OPTION-SETTING ROUTINES CANNOT BE EXECUTED UNDER SIMULATION, BUT MUST BE EXECUTED DIRECTLY, IN A TRAP (SEE BELOW). TO RELIEVE THE USER OF THE NECESSITY OF TRAPPING EVERY TIME AN OPTION IS TO BE CALLED, SOAPSUDS AUTOMATICALLY RECOGNIZES A CALL TO A SOAPSUDS OPTION, AND GENERATES A TRAP TO THE OPTION-SETTING ROUTINE INSTEAD OF SIMULATING THE CALL (THUS EXECUTING THE ROUTINE DIRECTLY). THE ORDINARY USER NEED NOT BE AWARE OF THIS MECHANISM, BUT MERELY KNOW THAT ANY SOAPSUDS OPTION CAN BE CALLED BEFORE INITIATING SOAPSUDS, UNDER SIMULATION, OR IN A TRAP ROUTINE. THE ONLY DIFFERENCE IN CALLING A SOAPSUDS OPTION BEFORE INITIALIZING SOAPSUDS IS THAT, IN THE EVENT OF AN IMPROPER PARAMETER IN THE CALL, NO MESSAGE WILL BE PRINTED. IN GENERAL, AN ERRONEOUS CALL WILL BE IGNORED, AND WILL NOT AFFECT FURTHER USE OF THE OPTION.

A NUMBER OF OTHER SUBROUTINES ARE INCLUDED IN THE LIST OF SUBPROGRAMS CALLS TO WHICH ARE TO GENERATE TRAPS. THIS LIST INCLUDES THE FORTRAN I-O ROUTINES, WHICH ARE OF COURSE NOT CODED TO PERMIT MANIPULATION OF THE BUFFERS BY SEVERAL PROCESSORS SIMULTANEOUSLY. OTHER ROUTINES, BOTH OF THE EXTERNAL OPERATING SYSTEM AND OF SUCH A MULTI-PROCESSOR OPERATING SYSTEM AS MAY BE USED, CAN BE INCLUDED. THE LIST IS IN SYMBLOIC (AS A LIST OF RETURN JUMPS) AT THE END OF THE SOAPSDS CONNECTING SUBROUTINE, AND SO MAY BE MODIFIED BY THE USER AS DESIRED.

USING THE TRAPPING AND TRACING SYSTEM

AS WAS MENTIONED ABOVE, SOAPSUDS IS PROVIDED WITH A SYSTEM FOR TRAPPING AND/OR TRACING OPCODES, LOADS, STORES, OR THE EXECUTION OF PARTICULAR INSTRUCTIONS.

ALL FOUR TYPES OF TRAPS WHICH ARE SET AND TURNED OFF AS DESCRIBED IMMEDIATELY BELOW ALWAYS TRAP BEFORE EXECUTING (I.E., SIMULATING) THE INSTRUCTION WHICH CAUSED THE TRAP. THUS, AN OPCODE TRAP, WHICH CAUSES A TRAP WHENEVER A PARTICULAR OPCODE IS EXECUTED BY A PARTICULAR CENTRAL PROCESSING UNIT, ALWAYS OCCURS BEFORE THE INSTRUCTION WITH THAT OPCODE IS EXECUTED. A CONTROL TRAP, WHICH CAUSES A TRAP WHENEVER AN INSTRUCTION AT A SPECIFIED LOCATION IS EXECUTED, ALWAYS OCCURS BEFORE ANY OF THE INSTRUCTIONS AT THAT LOCATION ARE EXECUTED (SIMULATED). LOAD AND STORE TRAPS, WHICH TRAP ON LOADS FROM OR STORES TO PARTICULAR LOCATIONS, ALSO OCCUR BEFORE THE LOAD OR STORE TAKES PLACE.

A FACILITY FOR TRAPPING ON A STORE AFTER THE STORE HAS TAKEN PLACE IS AVAILABLE THROUGH THE CHECK OPTION (SEE BELOW).

SETTING AND TURNING OFF TRAPS

TO TURN ON A TRAP,

```
CALL  TRAP(TYPE,ADDRESS,TRAPADR,CPU)
```

WHERE TYPE IS THE TYPE OF TRAP TO BE SET
=1 FOR AN OPCODE TRAP
=2 FOR A CONTROL TRAP
=3 FOR A LOAD TRAP
=4 FOR A STORE TRAP

ADDRESS IS THE NUMBER ON WHICH THE TRAP IS TO OCCUR
FOR AN OPCODE TRAP, THE OPCODE
FOR A CONTROL TRAP, THE LOCATION OF THE INSTRUCTION
FOR A LOAD TRAP, THE LOCATION FROM WHICH THE LOAD IS MADE
FOR A STORE TRAP, THE LOCATION TO WHICH THE STORE IS MADE

TRAPADR IS THE ADDRESS TO WHICH THE TRAP IS MADE WHEN THE TRAP CONDITION IS MET. SOAPSUDS PERFORMS A RJ TRAPADR WHEN IT TRAPS, SO THE LOCATION TRAPADR SHOULD BE BLANK, AND THE FIRST EXECUTABLE INSTRUCTION OF THE TRAP ROUTINE SHOULD BE AT TRAPADR+1.

CPU IS ONLY APPLICABLE ON OPCODE TRAPS. IT INDICATES THE NUMBER OF THE PROCESSOR TO WHICH THE OPCODE TRAP APPLIES.

ADDITIONAL NOTES

1. TO INDICATE 'EVERY' (EVERY OPCODE, EVERY ADDRESS, OR EVERY PROCESSOR), USE THE CONSTANT -64 (777677B). FOR EXAMPLE, TO TRAP TO ABSURD BEFORE THE EXECUTION OF EVERY INSTRUCTION,

```
CALL  TRAP(1,-64,ABSURD,-64)
```

2. ANY EVENT MAY CAUSE AT MOST ONE TRAP OF EACH TYPE. IF TWO TRAPS OF THE SAME TYPE ON THE SAME EVENT ARE SET, THE ONE SET FIRST IS THE ONE WHICH WILL BE EXECUTED. HOWEVER, AN 'EVERY' TRAP (A TRAP ON EVERY OPCODE FOR EVERY PROCESSOR, OR ON EVERY LOCATION) HAS PRECEDENCE OVER ALL OTHER TRAPS.

3. IT IS POSSIBLE FOR ONE INSTRUCTION TO LEAD TO SEVERAL TRAPS (THOUGH ONLY ONE OF EACH TYPE). IF AN INSTRUCTION CAUSES MORE THAN ONE TRAP, THE ORDER OF TRAPS WILL ALWAYS BE FIRST, CONTROL TRAP, SECOND, OPCODE TRAP, THIRD, LOAD OR STORE TRAP.

4. TRAPADR IS NOT CHECKED IN ANY WAY.

5. CPU NEED NOT BE GIVEN FOR TRAPS OTHER THAN OPCODE TRAPS.

TO TURN OFF A TRAP,

CALL TRAPOFF(TYPE,ADDRESS,0,CPU)

WHERE TYPE, ADDRESS AND CPU ARE THE SAME VALUES WHICH WERE USED IN CALLING TRAP WHEN THE TRAP WAS TURNED ON.
ADDITIONAL NOTES

1. AS FOR TRAP, B4 NEED NOT BE SET FOR TRAPS OTHER THAN OPCODE TRAPS.
2. IF TWO TRAPS HAVE THE SAME VALUES OF TYPE, ADDRESS (AND CPU, IF OPCODE TRAPS), THE FIRST ONE WHICH WAS SET WILL BE TURNED OFF.

THE USE OF TRAPS

TRAPS SERVE SEVERAL BASIC PURPOSES. THEY MAY BE USED TO COUNT THE FREQUENCY OF OPCODES, LOADS, OR STORES. TRAPS MAY BE USED TO TURN ON OR OFF OTHER OPTIONS AT PARTICULAR PLACES IN THE PROGRAM, OR TO CHECK THE VALUES OF SPECIAL LOCATIONS AT PARTICULAR INSTANTS. AND THEY MAY BE USED TO PRINT SPECIAL MESSAGES IN PLACE OF THE STANDARD TRACES IF DESIRED.

USING TRAP ROUTINES OFFERS SEVERAL ADVANTAGES. FIRST, CODING WHICH WOULD HAVE TO OCCUR AT A LARGE NUMBER OF POINTS IN A PROGRAM NEED NOT BE REPEATED (AS AN EXTREME EXAMPLE, TO DETERMINE OPCODE FREQUENCIES WITHOUT OPCODE TRAPS WOULD REQUIRE A ROUTINE AFTER EVERY INSTRUCTION OF THE PROGRAM). SECOND, SINCE TRAP ROUTINES ARE EXECUTED AND NOT SIMULATED, THEY ARE MUCH FASTER--IMPORTANT FOR LENGTHY PROCEDURES SUCH AS I-O. SINCE TRAP ROUTINES ARE NOT SIMULATED, THE MEMORY BOUNDS OF COURSE DO NOT APPLY TO THEM.

INFORMATION AVAILABLE DURING TRAPS

WHEN SOAPSUDS PERFORMS A TRAP, CERTAIN INFORMATION IS LEFT IN THE REGISTERS WHICH MAY BE USEFUL DURING THE TRAP ROUTINE. THE CONTENTS OF SOME OF THE REGISTERS UPON TRAPPING IS LISTED BELOW

IN B7 IS THE NUMBER OF THE PROCESSOR PRESENTLY EXECUTING, THE ONE WHICH CAUSED THE TRAP
IN B5 IS THE OPCODE OF THE INSTRUCTION PRESENTLY BEING EXECUTED
IN A1 IS THE INSTRUCTION LOCATION COUNTER OF THE PROCESSOR PRESENTLY EXECUTING
IN B1, FOR LOAD AND STORE TRAPS, IS THE LOCATION TO WHICH THE LOAD OR STORE IS TO BE MADE

EXITING FROM A TRAP ROUTINE

NORMAL EXIT FORM A TRAP ROUTINE IS JUST LIKE NORMAL EXIT FROM ANY SUBROUTINE, A JUMP TO THE FIRST WORD OF THE ROUTINE (TRAPADR). THIS WORD WILL CONTAIN A JUMP BACK INTO THE SOAPSUDS SIMULATION ROUTINES. IF A MESSAGE IS DESIRED AT THE END OF THE TRAP ROUTINE (THE SAME MESSAGE AS IS PRODUCED BY A TRACE) THE USER MUST RETURN TO THE LOCATION AFTER THE ONE IN THE JUMP INSTRUCTION AT TRAPADR. IN OTHER WORDS, HE MUST EXECUTE

```
SA1   TRAPADR
LX1   30
SB1   X1
JP    B1+1
```

SETTING AND TURNING OFF TRACES

AS WAS MENTIONED ABOVE, TRACES IN SOAPSUDS, WITH THE EXCEPTION OF OPCODE TRACES, ARE ACTUALLY SPECIAL TRAPS WHICH TRAP TO ROUTINES WHICH PRINT OUT THE REQUIRED TRACES. AS A RESULT, THE CALLING SEQUENCES FOR TURNING ON AND OFF TRACES ARE VERY SIMILAR TO THE SEQUENCES FOR TURNING ON AND OFF TRAPS.
TO TURN ON A TRACE,

```
CALL  TRACE(TYPE,ADDRESS,NAME,CPU)
```

WHERE TYPE, ADDRESS, AND CPU ARE THE SAME VALUES AS FOR A CALL TO TRAP, I.E.

TYPE IS THE TYPE OF TRACE TO BE SET
=1 FOR AN OPCODE TRACE
=2 FOR A CONTROL TRACE
=3 FOR A LOAD TRACE
=4 FOR A STORE TRACE

ADDRESS IS THE NUMBER ON WHICH THE TRACE IS TO OCCUR
FOR AN OPCODE TRACE, THE OPCODE
FOR A CONTROL TRACE, THE LOCATION OF THE INSTRUCTION
FOR A LOAD TRACE, THE LOCATION FROM WHICH THE LOAD IS MADE
FOR A STORE TRACE, THE LOCATION TO WHICH THE STORE IS MADE

CPU IS ONLY APPLICABLE ON OPCODE TRACES. IT INDICATES THE NUMBER OF THE PROCESSOR TO WHICH THE OPCODE TRACE APPLIES

NAME IS APPLICABLE TO CONTROL, LOAD, AND STORE TRACES. IT SHOULD BE THE ADDRESS OF A WORD WHICH CONTAINS, IN THE UPPER (LEFT) 48 BITS THE NAME OF THE LOCATION, IN DISPLAY CODE. THESE EIGHT CHARACTERS WILL BE PRINTED ALONG WITH THE CONTROL, LOAD, OR STORE TRACE PRINT-OUT. THE LOW-ORDER (RIGHTMOST) CHARACTER IN THE WORD SHOULD INDICATE, FOR LOAD AND STORE TRACES, THE FORMAT IN WHICH THE CONTENTS OF THE WORD BEING TRACED SHOULD BE PRINTED OUT. THUS, IF THE RIGHTMOST CHARACTER IS AN E, THE WORD WILL BE PRINTED OUT IN E FORMAT, IF AN F, IN F FORMAT, IF AN A, IN A10 FORMAT, IF AN L, IN L10 FORMAT, IF AN R, IN R10 FORMAT, AND IF AN I, IN I17 FORMAT. THE LETTER O AND MOST OTHER LETTERS WILL CAUSE PRINTOUT IN O20 FORMAT.

ADDITIONAL NOTES

1. THE INDICATION FOR 'EVERY' IS THE SAME AS IT IS FOR SETTING TRAPS, -64.

2. ANY EVENT MAY CAUSE EITHER A TRACE OR A TRAP OF ANY GIVEN TYPE, BUT NOT BOTH. TO BOTH TRACE AND TRAP, SET A TRAP AND PRINT A TRACE BY THE MEANS DESCRIBED UNDER 'EXITING FROM A TRAP', ABOVE. IF A TRACE AND A TRAP WERE SET ON A GIVEN LOCATION, THE ONE WHICH WAS SET FIRST WILL BE EXECUTED. HOWEVER, AN 'EVERY' TRAP OR TRACE HAS PRECEDENCE OVER ALL OTHER TRAPS AND TRACES.

3. ONE INSTRUCTION MAY PRODUCE SEVERAL TRAPS AND TRACES (THOUGH ONLY ONE OF EACH TYPE). THE ORDER IN WHICH THE TRAPS AND TRACES APPEAR IS THE SAME AS THE ORDER FOR A COMBINATION OF TRAPS, VIZ., FIRST, CONTROL TRAP OR TRACE, SECOND, OPCODE TRAP OR TRACE, THIRD, LOAD OR STORE TRAP OR TRACE.

4. ON CONTROL, LOAD, AND STORE TRAPS, THERE IS A LIMIT OF FIFTY TRACES AND TRAPS OF EACH TYPE. IF AT ANY TIME FIFTY TRACES AND TRAPS OF A TYPE ARE SET, ALL CALLS TO TRAP OR TRACE WILL BE IGNORED. THE LIMIT OF TWENTY ALSO APPLIES TO OPCODE TRAPS, BUT NOT TO OPCODE TRACES. OPCODE TRACES ARE HANDELED SPECIALLY, SO THAT THERE IS NO LIMIT ON THE NUMBER OF OPCODE TRACES WHICH MAY BE SET.

5. WARNING WHEN TRACE IS CALLED, SOAPSUDS RECORDS THE ADDRESS OF NAME, BUT DOES NOT COPY ITS CONTENTS INTO A REGION INSIDE SOAPSUDS. THUS,

IN ORDER TO PRESERVE THE NAME FOR FUTURE PRINT OUT, THE VARIABLE 'NAME' SHOULD NOT BE ALTERED AFTER THE CALL TO TRACE.

UNDER THE SOAPSUDS SYSTEM, TURNING OFF A TRACE IS LOGICALLY INDISTINGUISHABLE FROM TURNING OFF A TRAP. THE EXACT SAME ARGUMENTS TO TRAPOFF SHOULD BE USED AS WHEN TURNING OFF A TRACE. THUS, TO TURN OFF A TRACE,

CALL TRAPOFF(TYPE,ADDRESS,0,CPU)

WHERE THE VALUES OF TYPE, ADDRESS, AND CPU ARE THE SAME AS WERE USED IN TURNING ON THE TRACE. IF THERE ARE SEVERAL TRAPS AND/OR TRACES WITH THE SAME VALUES OF TYPE, ADDRESS (AND CPU), THE FIRST TRAP/TRACE WHICH WAS TURNED ON WILL BE THE ONE TURNED OFF.

TRACE FORMATS

THE FORMATS OF THE CONTROL, LOAD, AND STORE TRACES ARE QUITE SELF-EXPLANATORY. THEY ARE GIVEN BELOW FOR REFERENCE

STORE TRACE

```
PROCESSOR 23 AT P=007213 PERFORMED A STORE FROM X6 INTO ARRAY (013277)
     LOCN=7777777776777777777  (FORMERLY 0311170000000004321)
```

LOAD TRACE

```
PROCESSOR 23 AT P=007642 PERFORMED A LOAD FROM  BUFFER (002226) INTO
     X4                                          X4=0102240000000000000
```

CONTROL TRACE

```
PROCESSOR 23 HAS TRANSFERRED CONTROL TO LOCATION  LOOP2 (000212)
```

THE OPCODE TRACES, ON THE OTHER HAND, HAVE A NUMBER OF DIFFERENT FORMATS FOR THE DIFFERENT TYPES OF INSTRUCTIONS. ALL THE OPCODE TRACE FORMATS, HOWEVER, HAVE SOME COMMON CHARACTERISTICS. AT THE LEFT SIDE OF THE OUTPUT APPEAR THREE OCTAL NUMBERS. THE FIRST, A SIX DIGIT NUMBER, INDICATES THE ADDRESS OF THE INSTRUCTION PRESENTLY BEING EXECUTED. THE SECOND, A SINGLE OCTAL DIGIT, INDICATES THE QUARTER OF THE WORD IN WHICH THE OPCODE APPEARS. A 4 SIGNIFIES THE OPCODE IS IN THE HIGH ORDER BITS (LEFTMOST QUARTER OF THE WORD), AND A 1 SIGNIFIES THE OPCODE IS IN THE RIGHTMOST QUARTER OF THE WORD. THIRD APPEARS A TWENTY DIGIT NUMBER, THE WORD CONTAINING THE INSTRUCTION PRESENTLY BEING EXECUTED. AFTER THESE THREE NUMBERS APPEARS THE MNEMONICS FOR THE INSTRUCTION BEING EXECUTED. THEN APPEAR THE VALUES OF THE ENTRY OPERAND REGISTERS, AND AT THE RIGHT EDGE OF THE OUTPUT IS GIVEN THE RESULT OF THE OPERATION. FOR THE THREE INSTRUCTIONS WHICH HAVE TWO RESULT REGISTERS (OPCODES 24-26), ONE OF THE RESULT REGISTERS APPEARS IN THE OPERAND COLUMN. WHENEVER A SET A INSTRUCTION IS EXECUTED, THE CONTENTS OF THE ASSOCIATED X REGISTER IS ALSO PRINTED OUT. ABOVE EACH OPCODE TRACE APPEARS THE LINE

TRACE OF PROCESSOR NUMBER (ASSIGNED BADGENUMBER) 23

WHERE THE PROCESSOR NUMBER (HERE AND IN THE TRACES ABOVE) IS ALWAYS IN OCTAL. A SAMPLE OPCODE TRACE IS

```
 001071 4 61460072560612001077    SB4   B6+7256   B6=000072       B4=001171
```

IN GENERAL, IN SOAPSUDS TRACES, ADDRESSES AND THE K (CONSTANT) PORTION OF INSTRUCTIONS ARE ALWAYS WRITTEN OUT IN OCTAL. REGISTERS ARE ALWAYS

WRITTEN OUT IN OCTAL, AND ARE ALSO WRITTEN OUT IN INTEGER OR E FORMAT FOR INTEGER AND FLOATING ARITHMETIC INSTRUCTIONS RESPECTIVELY.

TRACING AND TRAPPING ON NO-OPS

A USER WILL OFTEN WISH A COMPLETE OPCODE TRACE (ALL OPCODES) OF A PARTICULAR PROCESSOR, BUT NOT WANT A TRACE OF EVERY NO-OP, WHICH WILL GENERALLY CONSTITUTE A CONSIDERABLE FRACTION OF THE OUTPUT, BUT ADD NO USEFUL INFORMATION. TO AVOID HAVING TO TURN OFF TRACES ON OPCODE 46 (NO-OP) EVERY TIME, AN OPTION HAS BEEN INCLUDED WICH WILL PERMANENTLY SUPPRESS TRACING OF AND TRAPPING ON NO-OPS. IT IS CALLED SIMPLY BY

CALL NONOOP

SUCCESSIVE CALLS TO NONOOP WILL TURN THE OPTION ALTERNATELY OFF AND ON.

SUSPENDING TRAPS AND TRACES

IN ORDER TO SUSPEND ALL TRAPS AND TRACES OF A PARTICULAR TYPE TEMPORARILY WITHOUT TURNING ALL THE TRAPS AND TRACES OFF, AND HAVING TO TURN THEM INDIVIDUALLY ON AGAIN, TWO OPTIONS HAVE BEEN PROVIDED IN SOAPSUDS, SUSPEND AND RESUME.

ALL TRAPS AND TRACES OF A GIVEN TYPE MAY BE SUSPENDED BY

CALL SUSPEND(TYPE)

WHERE TYPE IS THE TYPE OF TRAPPING AND TRACING TO BE SUSPNEDED
=1 FOR OPCODE TRACES AND TRAPS
=2 FOR CONTROL TRACES AND TRAPS
=3 FOR LOAD TRACES AND TRAPS
=4 FOR STORE TRACES AND TRAPS

THE CORRESPONDING CALL TO TURN BACK ON TRACES AND TRAPS OF A SPECIFIED TYPE AFTER THEY HAVE BEEN SUSPENDED IS

CALL RESUME(TYPE)

WHERE TYPE HAS THE SAME VALUE AS IT DID IN TURNING OFF THE TRAPS AND TRACES OF THAT TYPE.

IF NO TRAPS OR TRACES OF A GIVEN TYPE ARE SET WHEN A SUSPEND IS EXECUTED, THE SUSPEND WILL HAVE NO EFFECT. HOWEVER, EXECUTING A RESUME FOR A TYPE OF TRAP FOR WHICH NO TRAPS OR TRACES ARE SET WILL MAKE THE TRAPPING ROUTINE BELIEVE THAT IT ACTUALLY HAS SOME TRAPS SET, AND THUS CAN CAUSE CONFUSION

LOCATION CHECKING OPTIONS

AS WAS MENTIONED ABOVE, SOAPSUDS HAS TWO OPTIONS FOR THE CONSTANT MONITORING OF LOCATIONS, CHECK AND CORRECT. BOTH OPTIONS, ONCE TURNED ON, RECHECK THE VALUE OF THE SPECIFIED WORDS WHENEVER A STORE TO THOSE LOCATIONS IS MADE.

THE CHECK OPTION

THE CHECK OPTION AUTOMATICALLY CHECKS WHETHER A SPECIFIED LOCATION BEARS A SPECIFIED RELATION TO OTHER SPECIFIED LOCATION(S). IF THE RELATION IS NOT MET, TWO COURSES OF ACTION ARE AVAIABLE. EITHER A MESSAGE CAN BE PRINTED OUT, OR A TRAP CAN BE MADE TO A SPECIFIED LOCATION. UNLIKE A STORE TRAP, A CHECK TRAP IS MADE AFTER THE STORE OCCURS.
TO TURN ON A CHECK,

CALL CHECK(LOCN,NAME,REL,TRAPADR)

WHERE LOCN IS THE ADDRESS WHICH IS TO BE CHECKED
NAME IS THE ADDRESS OF A WORD IN WHOSE LEFT 48 BITS IS THE NAME OF THIS LOCATION, IN DISPLAY CODE. THESE EIGHT CHARACTERS WILL BE PRINTED OUT WHENEVER A CHECK MESSAGE FOR THIS LOCATION APPEARS. THE WORDS AT NAME+1 AND NAME+2 ARE USED IN COMPARISONS AGAINST THE CONTENTS OF LOCN
REL SPECIFIES THE RELATION TO BE TESTED FOR
THE RELATION IS
FOR REL=1, TRUE IF (LOCN).EQ.(NAME+1)
FOR REL=2, TRUE IF (LOCN).NE.(NAME+1)
FOR REL=3, TRUE IF (LOCN).GE.(NAME+1)
FOR REL=4, TRUE IF (LOCN).LT.(NAME+1)
FOR REL=5, TRUE IF (LOCN).GE.(NAME+1)
AND (LOCN).LT.(NAME+2) I.E., 'BETWEEN'
FOR REL=6, TRUE IF (LOCN).LT.(NAME+1) OR
(LOCN).GE.(NAME+1), I.E., 'NOT BETWEEN'
FOR REL=ANYTHING ELSE, AN ERROR MESSAGE WHEN CHECK IS CALLED
TRAPADR IS THE ADDRESS TO WHICH THE TRAP IS MADE IF THE RELATION IS NOT SATISFIED. AS WITH OTHER TRAPS, A RJ TRAPADR IS PERFORMED IN ORDER TO TRAP. HENCE THE FIRST EXECUTABLE STATEMENT OF THE TRAP ROUTINE SHOULD BE AT TRAPADR+1. EXIT FROM THE TRAP ROUTINE IS THE SAME AS EXIT FROM OTHER TYPES OF TRAPS (SEE ABOVE). IN THIS CASE, THE EXIT TO THE LOCATION AFTER THE ONE SPECIFIED IN THE JUMP STORED AT TRAPADR PRODUCES A 'CHECK MESSAGE.' IF TRAPADR IS ZERO NO TRAP WILL OCCUR, BUT THE 'CHECK MESSAGE' WILL BE PRINTED.

THE FORMAT OF THE 'CHECK MESSAGE' IS

```
PROCESSOR 23 AT P=001726 CHECKED VALUE OF LOCNAME (007555) AND FOUND THE
    ILLEGAL VALUE 000000000000000015B (                13)
```

THE CORRECT OPTION

THE CORRECT OPTION AUTOMATICALLY CHECKS WHETHER THE CONTENTS OF A SPECIFIED LOCATION IS EQUAL TO THE CONTENTS OF ANOTHER SPECIFIED LOCATION. WHENEVER A STORE IS MADE TO THE FIRST LOCATION. IF THE CONTENTS OF THE LOCATION BEING MONITORED IS SET TO A VALUE DIFFERENT FROM THE CONTENTS OF THE OTHER WORD (THE 'CORRECT VALUE'), A MESSAGE WILL BE PRINTED AND THE LOCATION BEING MONITORED IS RESET TO THE CORRECT VALUE.

THE CALLING SEQUENCE IS THE SAME AS FOR CHECK, EXCEPT B3 (REL) MUST BE NEGATIVE

```
CALL  CHECK(LOCN,NAME,-1)
```

WHERE LOCN IS THE ADDRESS WHICH IS TO BE CHECKED
NAME IS THE ADDRESS OF A WORD IN WHOSE LEFT 48 BITS IS THE NAME OF THIS LOCATION, IN DISPLAY CODE. THESE EIGHT CHARACTERS WILL BE PRINTED OUT WHENEVER A CORRECT MESSAGE FOR THIS LOCATION APPEARS. THE CONTENTS OF NAME+1 IS TAKEN AS THE 'CORRECT VALUE' WHICH IS COMPARED AGAINST LOCN WHENEVER A STORE IS MADE TO LOCN.

THE FORMAT OF THE CORRECT MESSAGE IS

```
PROCESSOR 23 AT P=001726 CORRECTED LOCNAME (007555) FROM
     0000000000000000013B (          11) TO 0000000000000000015B (      13)
```

THE PROCESSOR NUMBER AND ADDRESSES ARE IN OCTAL, AND THE CONTENTS ARE WRITTEN OUT IN BOTH OCTAL AND INTEGER FORMATS.

TO TURN OFF A CHECK OR A CORRECT

TO TURN OFF EITHER A CHECK OR A CORRECT OPTION SET ON A LOCATION,

```
CALL  CHECKOF(LOCN)
```

WHERE LOCN IS THE ADDRESS THE CHECK OR CORRECT OPTION ON WHICH IS TO BE TURNED OFF

ADDITIONAL NOTES

1. IF SEVERAL CHECKS AND/OR CORRECTS ARE SET ON ONE LOCATION, ONLY THE FIRST ONE SET WILL BE EXECUTED.
2. IF SEVERAL CHECKS AND/OR CORRECTS ARE SET ON ONE LOCATION, CHECKOF WILL TURN OFF THE ONE WHICH WAS TURNED ON FIRST.
3. A MAXIMUM OF 50 CHECKS AND CORRECTS MAY BE SET. WHEN 50 HAVE BEEN SET, FURTHER CALLS TO CHECK WILL BE IGNORED.
4. THE CHECK AND CORRECT OPTIONS MAY BE SUSPENDED AND RESUMED JUST LIKE THE TRAP OPTIONS, WITH CALLS TO SUSPEND AND TO RESUME. TYPE=5 FOR CALLS TO SUSPEND AND RESUME.

SUMMARY OF TRAP, TRACE, AND LOCATION CHECKING OPTIONS

THE FOLLOWING TABLE SUMMARIZES THE ARGUMENTS OF CALLS TO THE TRAPPING, TRACING, AND LOCATION CHECKING OPTIONS. FOR DETAILS SEE THE SPECIFIC DISCUSSIONS OF TRAPPING, TRACING, AND LOCATION CHECKING ABOVE.

OPTION	PAR1=	PAR2=	PAR3=	PAR4=	CALL
TO TURN ON OPCODE TRAP	1	OPCODE*	TRAPADR	CPU*	TRAP
TO TURN ON CONTROL TRAP	2	ADR*	TRAPADR	-	TRAP
TO TURN ON LOAD TRAP	3	ADR*	TRAPADR	-	TRAP
TO TURN ON STORE TRAP	4	ADR*	TRAPADR	-	TRAP
TO TURN ON OPCODE TRACE	1	OPCODE*	NAME	CPU*	TRACE
TO TURN ON CONTROL TRACE	2	ADR*	NAME	-	TRACE
TO TURN ON LOAD TRACE	3	ADR*	NAME	-	TRACE
TO TURN ON STORE TRACE	4	ADR*	NAME	-	TRACE
TO TURN ON A CHECK	LOCN	NAME	REL	TRAPADR	CHECK
TO TURN ON A CORRECT	LOCN	NAME	-1	-	CHECK
TO TURN OFF OPCODE TRAP	1	OPCODE*	-	CPU*	TRAPOFF
TO TURN OFF CONTROL TRAP	2	ADR*	-	-	TRAPOFF
TO TURN OFF LOAD TRAP	3	ADR*	-	-	TRAPOFF
TO TURN OFF STORE TRAP	4	ADR*	-	-	TRAPOFF
TO TURN OFF OPCODE TRACE	1	OPCODE*	-	CPU*	TRAPOFF
TO TURN OFF CONTROL TRACE	2	ADR*	-	-	TRAPOFF
TO TURN OFF LOAD TRAP	3	ADR*	-	-	TRAPOFF
TO TURN OFF STORE TRACE	4	ADR*	-	-	TRAPOFF
TO TURN OFF A CHECK	LOCN	-	-	-	CHECKOF
TO TURN OFF A CORRECT	LOCN	-	-	-	CHECKOF
TO SUSPEND OPCODE TRAPS AND TRACES	1	-	-	-	SUSPEND
TO SUSPEND CONTROL TRAPS AND TRACES	2	-	-	-	SUSPEND
TO SUSPEND LOAD TRAPS AND TRACES	3	-	-	-	SUSPEND
TO SUSPEND STORE TRAPS AND TRACES	4	-	-	-	SUSPEND
TO SUSPEND CHECKS AND CORRECTS	5	-	-	-	SUSPEND
TO RESUME OPCODE TRAPS AND TRACES	1	-	-	-	RESUME
TO RESUME CONTROL TRAPS AND TRACES	2	-	-	-	RESUME
TO RESUME LOAD TRAPS AND TRACES	3	-	-	-	RESUME
TO RESUME STORE TRAPS AND TRACES	4	-	-	-	RESUME
TO RESUME CHECKS AND CORRECTS	5	-	-	-	RESUME

* THE VALUE -64 MAY BE USED TO INDICATE 'ALL'
A DASH (-) INDICATES THE PARAMETER IS IGNORED BY THE ROUTINE.
THE OTHER ABBREVIATIONS ARE EXPLAINED IN THE DETAILED DESCRIPTIONS.

TIMING OPTIONS

THE SIMULATION OF INSTRUCTIONS BY SOAPSUDS MAKES THE ASSUMPTION THAT ALL INSTRUCTIONS, INCLUDING NO-OPS, REQUIRE THE SAME EXECUTION TIME. THIS EXECUTION TIME IS TERMED ONE CYCLE.

SOAPSUDS SIMULATES ONE INSTRUCTION FOR EACH PROCESSOR IN ONE SIMULATED CYCLE. THUS, IF THERE ARE THREE PROCESSORS RUNNING, SOAPSUDS WILL SIMULATE THE FIRST INSTRUCTION FOR PROCESSOR 0, THEN THE FIRST INSTRUCTION FOR PROCESSOR 1, THEN THE FIRST INSTRUCTION FOR PROCESSOR 2 (END OF FIRST CYCLE), THEN THE SECOND INSTRUCTION FOR PROCESSOR 0, ETC.

CYCLE TRACE

A COUNT OF THE PRESENT CYCLE NUMBER IS KEPT IN SOAPSUDS (FIRST CYCLE= CYCLE 0). THE USER MAY OPTIONALLY PRINT OUT AT THE END OF EACH CYCLE A MESSAGE OF THE FORM

```
CYCLE 1239 JUST COMPLETED
```

WHERE THE NUMBER OF THE CYCLE IS IN DECIMAL.

THIS OPTION IS TURNED ON BY

```
CALL CYCLTRC (1)
```

AND TURNED OFF BY

```
CALL CYCLTRC (0)
```

CYCLE TRAPS

THE USER MAY CAUSE A TRAP TO BE EXECUTED AT A PARTICULAR CYCLE COUNT, THEREBY ENABLING HIM, FOR EXAMPLE, TO SET A TRACE JUST BEFORE THE PROGRAM ENCOUNTERS DIFFICULTIES.

TO TRAP ON A CYCLE COUNT,

```
CALL CYCTRP (CYCLE,LOCN)
```

WHERE CYCLE IS THE CYCLE COUNT TO BE TRAPPED ON AND LOCN IS THE LOCATION TO WHICH THE TRAP IS MADE.

THUS, TO TRACE ALL THE INSTRUCTIONS EXECUTED BETWEEN CYCLES 100000 AND 110000, ONE WOULD DO THE FOLLOWING

```
EXTERNAL CN,CF
CALL CYCTRP (100000,CN)
CALL CYCTRP (110000,CF)

SUBROUTINE CN
CALL TRACE (1,-64,0,-64)
RETURN
END

SUBROUTINE CF
CALL TRAPOFF (1,-64,0,-64)
RETURN
END
```

TIME STATUSES

IN ADDITION TO KEEPING A COUNT OF THE PRESENT CYCLE NUMBER, SOAPSUDS KEEPS FOUR COUNTS FOR EACH OF THE PROCESSORS--SYSTEM TIME, PROGRAM TIME, IDLE TIME, AND TRACING TIME, ALL OF WHICH ARE GIVEN IN CYCLES. NORMALLY THE RUNNING TIME IS INCREMENTED EACH CYCLE SO LONG AS THE PROCESSOR IS RUNNING, AND THE IDLE TIME IS INCREMENTED EACH CYCLE ONCE THE PROCESSOR PROCESSOR HAS STOPPED.

BY USE OF THE SYSTEMS, USEFUL, USELESS, AND TRACING OPTIONS THE USER CAN OVERRIDE THIS CONVENTION AND HAVE ANY OF THE TIME COUNTERS INCREMENTED, REGARDLESS OF THE STATE OF THE PROCESSOR. THESE OPTIONS ARE INTENDED TO GIVE USERS THE POSSIBILITY OF DISCRIMINATING BETWEEN TIME SPENT IN CONTROLLING THE THE OVERALL FLOW OF PROGRAM EXECUTION , IN THE ACTUAL EXECUTION OF PROGRAMS, IN NONPRODUCTIVE ACTIVITY AND IN THE PERFORMANCE OF MISCELLANEOUS TASKS RESPECTIVELY.

THE STAUSES ARE SET AS FOLLOWS--

SYSTEM TIME STATUS	CALL SYSTEMS
PROGRAM TIME STATUS	CALL USEFUL
IDLE TIME STATUS	CALL USELESS
TRACING TIME STATUS	CALL TRACING

THESE OPTIONS IN NO WAY AFFECT SIMULATION. REPEATED CALLS TO THEM WILL NOT CAUSE ANY HARM.

SOAPSNAP

WHEN ALL PROCESSORS HAVE STOPPED, AT THE END OF SIMULATION, SOAPSUDS PRINTS OUT A 'SOAPSNAP', WHICH GIVES THE PRESENT STATUS OF ALL PROCESSORS (IN THIS CASE, ALL STOPPED) AND THE RUNNING AND IDLE TIMES OF ALL THE PROCESSORS. SUCH A SOAPSNAP MAY OPTIONALLY BE PRINTED OUT AT ANY TIME DURING SIMULATION, IF, FOR EXAMPLE, THE TIME REQUIRED FOR INDIVIDUAL ROUTINES IS BEING MEASURED.

TO SOAPSNAP WHILE IN A TRAP ROUTINE OR UNDER SIMULATION,

```
CALL    SOAPSNP
```

OR, FOR EXAMPLE, TO SOAPSNAP ON EXITING FROM A SUBROUTINE CALLED SPS,

```
CALL    TRAP (2,SPS,SOAPSNP)
```

SNAP

THE USER MAY PRINT OUT A SNAP OF THE CURRENT REGISTER INFORMATION OF ANY CPU BY A CALL TO SNAP. IN ADDITION TO THE CONTENTS OF THE X,A AND B REGISTERS, THE CURRENT P REGISTER, INSTRUCTION, STATUS (RUNNING OR STOPPED), MODE (PROGRAM OR RESIDENT), CYCLE COUNT AND THE TOTAL TIME SPENT IN EACH OF THE TIME STATUSES ARE PRINTED OUT.

SOAPSUDS ALWAYS PRINTS A SNAP ON THE DETECTION OF AN ERROR CONDITION AFTER THE PRINTOUT DESCRIBING THE CONDITION. TO SNAP,

```
CALL    SNAP
```

ABORT OPTION

IF THE USER HAS ERROR-CHECKING ROUTINES IN THE PROGRAM BEING SIMULATED, AND HE DETECTS AN ERROR, HE MAY WISH TO ABORT HIS PROGRAM RATHER THAN END IT (I.E., CALL EXIT). SOAPSUDS HAS AN OPTION WHICH WILL PRINT OUT THE MESSAGE

$$$$$ EXECUTION TERMINATED BY ABORT REQUESTED BY PROGRAM (DURING SIMULATION OF CPU 23, P=007327) DURING CYCLE 1324

AND THEN ABORT THE PROGRAM. THE CPU NUMBER AND P REGISTER ARE PRINTED IN OCTAL, AND THE CYCLE COUNT IN DECIMAL, AS USUAL.

TO ABORT, FOR EXAMPLE, IF CONTROL PASSES TO LOCATION ILLEGAL, PUT AT LOCATION ILLEGAL

```
        CALL    ABORT
```

OR

```
        CALL TRAP (2,ILLEGAL,ABORT,0)
```

SUMMARY OF MISCELLANEOUS SOAPSUDS FEATURES

THE FOLLOWING TABLE SUMMARIZES THE ARGUMENTS OF CALLS TO VARIOUS FUNCTIONS OF THE SOAPSUDS SYSTEM. AGAIN, FOR DETAILS SEE THE SPECIFIC DISCUSSIONS OF THE FUNCTIONS IN THE TEXT.

FUNCTION	PAR1	PAR2	PAR3	PAR4	CALL
TO PRINT OUT CYCLE TRACE	1	-	-	-	CYCLTRC
TO TURN OFF CYCLE TRACE	0	-	-	-	CYCLTRC
TO TURN ON A CYCLE TRAP	CYCLE	LOCN	-	-	CYCTRP
TO START IDLE TIME INCR	-	-	-	-	USELESS
BY SETTING TRAP*	TYPE	ADR	USELESS	(CPU)	TRAP
TO START RUNNING TIME INCR	-	-	-	-	USEFUL
BY SETTING TRAP*	TYPE	ADR	USEFUL	(CPU)	TRAP
TO START SYSTEMS TIME INCR	-	-	-	-	SYSTEMS
BY SETTING TRAP*	TYPE	ADR	SYSTEMS	(CPU)	TRAP
TO START TRACING TIME INCR	-	-	-	-	TRACING
BY SETTING TRAP*	TYPE	ADR	TRACING	(CPU)	TRAP
TO SOAPSNAP DURING TRAP OR SIMULATION	-	-	-	-	SOAPSNP
BY SETTING TRAP*	TYPE	ADR	SOAPSNP	(CPU)	TRAP
TO SNAP DURING TRAP OR SIMULATION	-	-	-	-	SNAP
BY SETTING TRAP*	TYPE	ADR	SNAP	(CPU)	TRAP
TO ABORT DURING TRAP OR SIMULATION	-	-	-	-	ABORT
BY SETTING TRAP*	TYPE	ADR	ABORT	(CPU)	TRAP

* THESE ENTRIES ARE MERELY TO INDICATE THAT, SINCE THESE SUBROUTINES REQUIRE NO ARGUMENTS, THEY MAY BE CALLED DIRECTLY UPON TRAPPING WITHOUT THE USE OF AN INTERMEDIATE TRAP ROUTINE TO CALL THE OPTION. THEY MAY BE CALLED BY ANY OF THE FOUR STANDARD TYPES OF TRAPS (FOR A DESCRIPTION OF WHICH, SEE 'USING THE TRAPPING AND TRACING SYSTEM', ABOVE). THEY MAY ALSO BE CALLED FROM CHECK TRAPS (DESCRIBED UNDER 'LOCATION CHECKING OPTIONS', ABOVE).

APPENDIX I

PLANS FOR THE FUTURE

IT SHOULD BE CLEAR FROM THE DESCRIPTION OF THE SYSTEM GIVEN IN THE TEXT THAT SOAPSUDS WILL BE A USEFUL AND FLEXIBLE TOOL FOR EXPERIMENTATION WITH PARALLEL PROCESSING PROCEDURES. THERE ARE, HOWEVER, SEVERAL ASPECTS OF SUCH PROCEDURES THAT ARE NOT BROUGHT OUT CLEARLY ENOUGH BY THE PRESENT SYSTEM, THESE INCLUDE A MEANS OF RECORDING THE SIMULATANEOUS USAGE OF FUNCTIONAL HARDWARE UNITS AND FAST REGISTERS, MEMORY CONFLICTS, AND AN ACCURATE ESTIMATE OF THE IDLING TIME VS. RUNNING TIME RELATIONSHIP. IN ADDITION, CERTAIN FEATURES SHOULD BE ADDED TO THE SOAPSUDS REPERTOIRE TO ENABLE A STUDY OF THE FEASIBILITY AND DESIRABILITY OF VARIOUS POSSIBLE SYSTEM CONFIGURATIONS. SHOULD ONE, FOR EXAMPLE, SELECT A PARTICULAR CPU AS AN EXECUTIVE CONTROLLING THE OPERATION OF THE OTHER PROCESSORS. SHOULD THIS SELECTION BE FLEXIBLE TO ALLOW SHARING OF SYSTEM FUNCTIONS AMONG SEVERAL CPU'S AS REQUIRED. SHOULD THE SYSTEM BE SET UP SO THAT AS MUCH AS POSSIBLE EACH CPU TAKES CARE OF ITS OWN SYSTEM FUNCTIONS. FOR THESE REASONS. AN AUGMENTED VERSION OF SOAPSUDS IS BEING DESIGNED SOME CHARACTERISTICS OF WHICH WILL NOW BE OUTLINED.

I IT IS INTUITIVELY CLEAR THAT A PARALLEL PROCESSOR WILL NOT REQUIRE A FULL SET OF FUNCTIONAL UNITS (I.E. THE HARDWARE FOR MULTIPLICATION, DIVISION, ETC.) FOR EACH CPU. IN ORDER TO GET A CONCRETE PICUTRE OF THE AMOUNT OF SHARING THAT CAN EFFICIENTLY BE TOLERATED, A FUNCTIONAL UNIT USE ANALYZER WILL BE INCORPORATED INTO THE SYSTEM ALONG WITH THE ABILITY TO VARY THE NUMBER OF FUNCTIONAL UNITS TO BE ALLOWED (AND A DETERMINATION OF THE LENGTHS OF ANY RESULTING QUEUES).

II THE AUGMENTED SYSTEM WILL DISPLAY THE AMOUNT OF TIME LOSS CAUSED BY CONFLICTS IN MEMORY REFERENCING, THIS FEATURE WILL BE SO WRITTEN AS TO ALLOW VARIATIONS IN THE DEFINITION OF ''MEMORY CONFLICT.''

III IT WOULD BE DESIRABLE TO MEASURE THE USEFULNESS OF HAVING A POOL OF FAST REGISTERS THAT ALL THE CPU'S COULD DRAW FROM RATHER THAN A FIXED CONTINGENT FOR EACH CPU, A SCHEME FOR ACCOMPLISHING A REASONABLE ESTIMATE FOR THIS WILL BE ADDED TO THE AUGMENTED SYST.

IV SOAPSUDS CURRENTLY DOES ONLY A ROUGH ESTIMATE OF THE COMPUTER TIME USED BY THE CPU'S. THE AUGMENTED VERSION WILL MORE TRULY REFLECT THIS AND ALLOW REDEFINITION OF INSTRUCTION TIMES FOR EXPERIMENTATION.

V IN ORDER TO FACILITATE EXECUTIVE CONTROL OF THE OPERATIONS OF THE SYSTEM. AN EXCHANGE JUMP* INSTRUCTION WILL BE INCORPORATED IN THE NEW SYSTEM.

VI FINALLY, THE STORAGE PROTECTION FEATURES OF SOAPSUDS WHICH ARE QUITE LIMITED AT PRESENT WILL BE AUGMENTED BY A MORE FLEXIBLE SET UP.

*CF. 6600 REF. MANUAL 3-9

APPENDIX II

ATHENA, AN OPERATING SYSTEM FOR SOAPSUDS

(E. DRAUGHON SYSTEM)

AS PART OF THE EXPERIMENTATION WITH SOAPSUDS, PLANS HAVE BEEN MADE FOR A POSSIBLE OPERATING SYSTEM FOR A PARALLEL MULTIPROCESSOR CONFIGURATION LIKE THAT SIMULATED BY SOAPSUDS. TESTING OF THIS OPERATING SYSTEM UNDER AN EXTENDED VERSION OF SOAPSUDS WILL YEILD DATA ON THE FEASIBILITY AND EFFICIENCY OF VARIOUS TECHNIQUES REQUIRED FOR THE ORGANIZATION AND CONTROL OF A MULTIPROCESSOR SYSTEM.

THE OPERATING SYSTEM DESCRIBED BELOW, ATHENA, USES THE PRINCIPLE OF TASKING IN ORDER TO SUBDIVIDE THE JOBS BEING RUN TO PERMIT PARALLEL PROCESSING. A TASK IS THE EXECUTION OF A PARTICULAR SECTION OF CODE BY ONE PROCESSOR. NOTE THAT THERE IS NOT A ONE-TO-ONE CORRESPONDENCE OF TASKS AND SECTIONS OF CODE--ANY NUMBER OF TASKS MAY ALL REFER TO THE SAME SECTION OF THE PROGRAM. IN A VERY LARGE DO LOOP, FOR EXAMPLE, EACH PASS THROUGH THE DO LOOP (I.E., EACH VALUE OF THE ITERATION VARIABLE) MAY BE MADE INTO A SEPARATE TASK. THE BASIC FACT ABOUT A TASK IS THAT, ONCE IT BEGINS EXECUTION, IT DOES NOT WAIT FOR SYSTEM APPROVAL TO EXECUTE ANY SUBSECTION OF THE TASK. THAT IS, AS FAR AS THE SYSTEM IS CONCERNED, IT INITIATES EXECUTION OF THE TASK ONLY WHEN IT IS ALL RIGHT FOR THE TASK TO EXECUTE TO COMPLETION. THE TASKS BELONGING TO A PARTICULAR JOB CAN, OF COURSE, STILL COMMUNICATE WITH EACH OTHER THROUGH STORAGE FLAGS, SO THAT ONE TASK MAY PAUSE IN THE MIDDLE OF ITS CODE WAITING FOR A PARTICULAR FLAG TO BE SET IN MEMORY. HOWEVER, UNLESS THE EXPECTED WAIT IS VERY SHORT, SUCH PROCEDURES DEFEAT THE PRUPOSE OF A TASKING SYSTEM, SINCE THE IDEA IS TO EXECUTE FIRST, AT ANY GIVEN TIME, THOSE SECTIONS OF CODE WHICH MAY BE PROCESSED TO COMPLETION WITHOUT PAUSE. THIS MINIMIZES THE TIME THAT THE CENTRAL PROCESSORS ARE WAITING IDLY IN TASKS, AND THUS PRESUMABLY MAXIMIZES THE COMPUTING POWER OF THE MACHINE.

WHETHER THIS SYSTEM IS IN FACT THE MOST EFFICIENT WILL DEPEND LARGELY ON THE PROPROTION OF TIME REQUIRED FOR SYSTEM OVERHEAD. AFTER THE COMPLETION OF EACH TASK, THE PROCESSOR RETURNS TO A SYSTEM ROUTINE KNOWN AS THE RESIDENT. IT IS THE JOB OF THE RESIDENT TO DETERMINE WHAT TASK , IF ANY, IS READY TO BE EXECUTED. IF IT FINDS ONE, THE PROCESSOR TRANSFERS CONTROL TO THIS TASK. IN ADDITION TO THE RESIDENT EXECUTED BY EVERY PROCESSOR, AT LEAST ONE PROCESSOR MUST BE SET ASIDE TO COORDINATE THE ENTIRE SYSTEM AND REGULATE OVERALL JOB THROUGHPUT.

BELOW IS A DESCRIPTION OF THE SPECIFIC ROUTINES IN THE ATHENA SYSTEM, AND THE FORTRAN SUBROUTINES WHICH HAVE BEEN WRITTEN TO MAKE THE SYSTEM EASILY USABLE.

```
.....................ATHENA.........................
.
. LISTS.
.     INPUT LIST.  LIST OF EACH MAIN PROGRAM OR JOB IN THE MACHINE.
.     JOB LIST.  LIST OF ALL THOSE MAIN PROGRAMS THAT CURRENTLY
.           HAVE TASKS TO BE PERFORMED.
.     TASK LISTS.   ONE LIST FOR EACH MAIN PROGRAM OR JOB.
.              EACH LIST LISTS ALL THE TASKS WHICH THAT MAIN
.                   PROGRAM HAS CURRENTLY TO BE PERFORMED.
.
.
. METHOD.
.   COORDINATOR.
```

1. KEEPS THE JOB LIST UP TO DATE BY LOOKING AT EACH TASK LIST IN THE SYSTEM. IF THE TASKLIST IS EMPTY, THE JOB IS TAKEN OFF THE JOB LIST, OTHERWISE THE JOB REMAINS ON THE JOB LIST OR IS ADDED TO THE JOB LIST.

RESIDENT.

1. WAITS FOR A JOB TO APPEAR ON THE JOB LIST.
2. WHEN IT DOES, THE JOB'S TASKLIST IS SEARCHED. IF ANY TASKS REMAIN TO BE DONE, THE PROCESSOR EXITS AND DOES THE TASK. OTHERWISE, IT RETURNS TO LOOK FOR THE NEXT JOB IN THE JOBLIST.
3. WHEN A RELEASE IS MADE, THE PROCESSOR EXITS TO SYSTEM STATUS, ADDS THE TASK TO THE TASKLIST FOR THIS JOB AND EXITS BACK TO PROGRAM STATUS AND CONTINUES EXECUTING THE PROGRAM.
4. WHEN A RETURN IS ENCOUNTERED, THE PROCESSOR EXITS TO SYSTEM STATUS AND BEGINS SEARCHING THE TASKLIST AGAIN (AS IN 2. ABOVE).

III. USAGE.

1. TO HANDLE A TASK WITH ARGUMENTS...

SUPPOSE THAT THE TASK IS CALLED SUB AND ONE WANTS TO START IT 3 TIMES SIMULTANEOUSLY WITH DIFFERENT ARGUMENTS.

```
      CALL   RELEASE (14S,3,6S)
   14 GO TO 21
      GO TO 22
      GO TO 23

   21 CALL   SUB(ARG1,ARG2, ... ,ARGN)
   22 CALL   SUB(ARG1,ARG2, ... ,ARGN)
   23 CALL   SUB(ARG1,ARG2, ... ,ARGN)


   6  CONTINUE
```

2. TO HANDLE PRINT STATEMENTS PRINTED BY MORE THAN ONE PROCESSOR.....

```
       CALL  SKIPPER(0)

       PRINT 5
    5  FORMAT ( * THIS IS A PRINT STATEMENT WITHIN THE PROG
   X  RAM TO BE EXECUTED BY ANY NUMBER OF PROCESSORS.*)

       CALL   SKIPPER(1)
```

3. TO GIVE UP A PROCESSOR TO THE SYSTEM....

```
       CALL   RETURN
```

4. TO RELEASE A NUMBER OF TASKS TO BEGIN EXECUTING AS SOON AS POSSIBLE...

```
CALL   RELEASE(LIST,N,CONTIN)
```

(IN THIS CALL, LIST IS THE BEGINNING ADDRESS OF A LIST OF TASK ADDRESSES. THE ADDRESSES ARE TO BE RIGHT JUSTIFIED IN THE LEFT HALF OF THE WORD, SUCH AS WOULD BE GENERATED BY A GO TO STATEMENT IN FORTRAN. N IS THE NUMBER OF TASKS BEING RELEASED. CONTIN IS THE STATEMENT NUMBER OF THE NEXT EXECUTABLE INSTRUCTION. AFTER THE CALL, OR THE ADDRESS OF SUCH.)

(IF THE VALUE OF N IS -1, AS MANY PROCESSORS AS ARE AVAILABLE WILL BEGIN EXECUTING THE TASK. OTHER PROCESSORS WILL JOIN IN EXECUTING THE TASK UNTIL A CALL TAKEOFF IS EXECUTIED.)

5. TO PREVENT FURTHER PROCESSORS FROM ENTERING A TASK AT WILL, EXECUTE A ...

```
CALL   TAKEOFF(ADD,CONTIN)
```

(ADD IS THE ADDRESS OF THE TASK, CONTIN THE NEXT EXECUTABLE INSTRUCTION.)

6. TO RETURN ALL PROCESSORS BUT ONE TO THE SYSTEM...

```
CALL   IFTHRU (KALLIES,N,CONTIN)
```

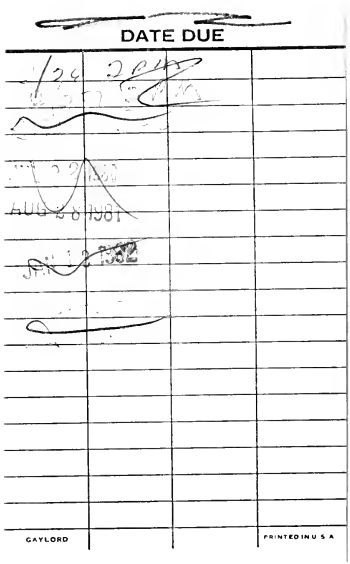
(KALLIES IS AN OTHERWISE UNUSED LOCATION. N IS TGE NUMBER OF TASKS WHICH ARE TO BE FINISHED BEFORE THE NEXT STATEMENT (AT ADDRESS CONTIN) IS EXECUTED.)

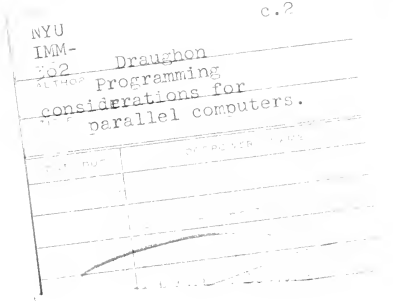
7. TO HANDLE PARALLEL DO LOOPS....

```
CALL   DO (QK,K,INC,LIM,CONTIN,NUMB)
```

(QK IS THE PUBLIC ITERATION VARIABLE. K IS THE PRIVATE ITERATION VARIABLE. INC IS THE AMOUNT OF INCREMENT. LIM IS THE LIMITING VALUE OF THE ITERATION VARIABLE. NUMB IS THE NUMBER OF PROCESSORS TO BEGIN THE DO LOOP. THE ITERATION VARIABLE QK MUST BE INITIALIZED BEFORE THE CALL TO THIS ROUTINE. THE CALL ABOVE REPLACES THE DO STATEMENT WHICH IS TO BE PARALLEL.)

.
.
.
.
.....


DATE DUE
GAYLORD
PRINTED IN U S A



N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012